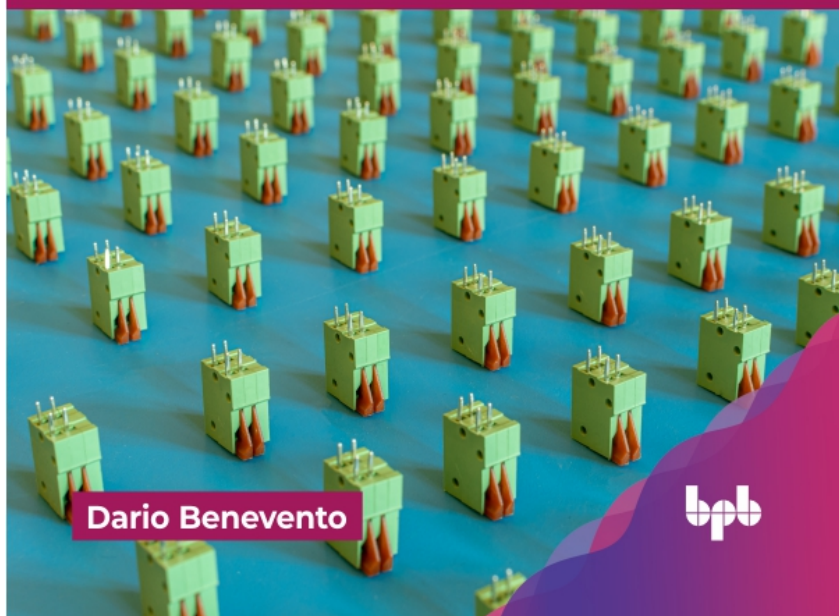


Architecting **ASP.NET Core** Applications

ASP.NET Core backend with C# 13 and .NET 9



Dario Benevento



Architecting ASP.NET Core Applications

*ASP.NET Core backend with
C# 13 and .NET 9*

Dario Benevento



www.bpbonline.com

First Edition 2025

Copyright © BPB Publications, India

ISBN: 978-93-65899-917

All Rights Reserved. No part of this publication may be reproduced, distributed or transmitted in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication, photocopy, recording, or by any electronic and mechanical means.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

**To View Complete
BPB Publications Catalogue
Scan the QR Code:**



www.bpbonline.com

Dedicated to

*In loving memory of my mother,
who always encouraged me to study;
My father,
who always believed in me;
My wife **Raffaella**,
without whose help this book wouldn't have been possible;
And my sons **Thomas** and **Julian**,
from whom I stole precious time.*

About the Author

Dario Benevento is a senior software developer at a leading Swiss company in the machine tool sector, currently responsible for developing ASP.NET Core and the graphical interface with Blazor. With 40 years of software experience, including 30 professionally, he specializes in Microsoft technologies, software design, and system architecture. His curiosity, precision, and ability to adapt quickly to changes have elevated his professionalism to an excellent level.

For many years, he has participated in community events, and more recently, he has begun sharing his knowledge with others as a speaker at conferences on Blazor and .NET, thus sharing his passion and expertise. Writing this book represents another step in his journey of knowledge sharing.

About the Reviewers

- **Alberto Mori** is a software developer with a deep passion for web technologies and software development, specializing in .NET. Currently, he works as a software developer at Able Tech srl.

Actively engaged in the tech community, Alberto is the host of the online show Spike Time for UGIdotNET, the .NET Italian User Group, where he explores new technologies in each episode. He shares his experience by speaking at conferences and community events.

As a testament to his expertise and dedication, Alberto has been awarded the prestigious title of Microsoft MVP, recognizing his contributions to the global developer community and his commitment to fostering collaboration and innovation.

- **Andrea** is a .NET developer with a degree in computer engineering and experience in building web and desktop applications using Blazor and WPF, with a focus on creating responsive and secure solutions. Skilled in C# and object-oriented programming, he also has hands-on experience with backend technologies such as SQL Server and Entity Framework. Currently, he works as a consultant for a company with a focus in communications security and data encryption. He is a Linux enthusiast and passionate about clean code and scalable solutions.

Acknowledgement

Writing this book has been a journey of growth, discovery, and collaboration, and I would like to express my deepest gratitude to everyone who helped me complete it.

First, I would like to thank my family. My mother, who unfortunately will never see this book completed, always encouraged me to study and was truly a guiding light in my life. My father, whose unwavering belief in me has always been a source of strength. My wife, Raffaella, without whose patience, understanding, and support this book would not have been possible. And my sons, Thomas and Julian, who generously sacrificed their time with me so I could focus on this project. Your love and support mean everything to me.

I extend my gratitude to my colleagues and mentors who have inspired me throughout my career. Lorenzo Crivelli has always guided me toward the right professional choices; debates with Cem Soyding allowed me to deepen many of the topics in this book; Francesco Baderna, who first introduced me to object-oriented programming (feels like ages ago, doesn't it?); and my cousin Roberto Bruno, also a software developer—even better than me—who has always encouraged and supported me since the beginning of this journey.

Special thanks to the technical reviewers:

- **Alberto Mori**, a friend and community companion who significantly contributed to this book by pointing out errors and inaccuracies in both the code and text.
- **Andrea Graziani**, a friend, former colleague, and great theorist, who also corrected many inaccuracies and concepts that I took for granted but were not so obvious to an attentive reader like him.

To the editors who dedicated their time to ensuring the clarity and

quality of this book. Your expertise has significantly improved this work.

Finally, I want to thank the *Blazor Developers Italiani* community, whose passion for learning and innovation continuously inspires me. This book is a testament to my desire to give back and share the knowledge I have gained over the years.

Thank you all for being part of this journey.

Preface

In today's software development world, understanding the principles of software architecture has become essential. This book, *Architecting ASP.NET Core Applications*, is designed to provide a clear and practical guide to building scalable, secure, and maintainable software systems.

The book is structured into carefully designed chapters covering a wide range of fundamental topics for modern software development. Starting from the basics, we explore foundational design principles and key architectural patterns, delve into the world of RESTful APIs, dependency injection, middleware, and advanced design practices such as CQRS and modular monoliths.

Additionally, we will cover cutting-edge technologies such as containerization with Docker, orchestration with Kubernetes, and real-time communication with SignalR. The book also dedicates significant attention to security, testing, and performance optimization, ensuring applications meet the most modern demands for performance, maintainability, and lifecycle management.

By mastering the concepts and techniques presented in this book, you will be able to design and implement robust software systems that stand the test of time. This book will undoubtedly serve as a valuable companion in every developer's professional journey to mastering software architecture with ASP.NET Core.

Chapter 1: Introduction to ASP.NET Core - This chapter introduces ASP.NET Core, providing an overview of its structure, functionality, and complexity. We begin with installing Visual Studio and selecting options to develop ASP.NET Core applications. The chapter also delves into fundamental software development principles like SOLID, KISS, and DRY, highlighting their importance in building robust and maintainable applications. Starting with a solid foundation is the key to tackling the challenges of complex

software projects.

Chapter 2: Basics of ASP.NET Core - This chapter explores the structure of an ASP.NET Core project and the functioning of middleware and the request pipeline. It provides a practical introduction to basic design patterns, explaining how to apply them to enhance the scalability and maintainability of applications.

Chapter 3: Architecture and Core Components - This chapter offers an overview of major software architectures, including MVC, Hexagonal Architecture, Clean Architecture, and Domain-Driven Design. Each approach is analyzed in the context of ASP.NET Core, highlighting its benefits and ideal use cases. This chapter helps in selecting the most suitable architecture for complex projects.

Chapter 4: Designing RESTful APIs - This chapter explains REST and RESTful APIs, providing a practical guide to using ASP.NET Core with both classic WebAPIs (controller-based) and minimal APIs. It explores structure, functionality, and best practices for designing efficient, secure, and high-performing APIs, ensuring their integration into modern systems.

Chapter 5: Implementing Routing and in ASP.NET Core - The chapter introduces routing concepts in ASP.NET Core, explaining how to map URLs to controllers and actions. It explores configuration and customization options, focusing on parameters, constraints, and best practices for effective and flexible routing in complex applications.

Chapter 6: Middleware and Extensibility - This chapter examines dependency injection concepts and middleware functionality in ASP.NET Core. It demonstrates how to create custom middleware to tailor the application to specific needs and extend dependency injection, integrating modular components to improve flexibility and extensibility.

Chapter 7: Architectural Principles - This chapter addresses backend architectural principles, focusing on functionality-oriented techniques and the connection between SOLID principles and software architecture. It provides tools for structuring robust and scalable applications.

Chapter 8: GoF Design Patterns - This chapter analyzes the

application of some design patterns from the GoF book in ASP.NET Core, such as Singleton, Factory Method, Decorator, Observer, and Strategy. Each pattern is explained with practical examples, showcasing how to apply them to solve common problems in daily software development.

Chapter 9: CQRS in Architecture - The chapter introduces the concept of CQRS, delving into the separation between commands and queries and integration with Event Sourcing. It provides a practical guide to implementing CQRS in real-world projects, improving code organization.

Chapter 10: Modular Monolith - The goal of this chapter is to explore modular design in ASP.NET Core, analyzing the structure of this project type, communication between modules, and dependency management. It also includes insights into testing, logging, monitoring, and deployment to ensure efficient management of modular monoliths.

Chapter 11: SignalR in Real-time Applications - This chapter describes SignalR and its use in .NET9 for real-time communication between backend and frontend. It explains how to send messages to individual clients or groups, ensuring application security and optimizing interactivity in complex scenarios.

Chapter 12: Automated Testing - This chapter emphasizes the importance of automated testing, exploring different test types such as unit, integration, and black-box. It provides practical guidelines for implementing effective tests, improving the quality and reliability of ASP.NET Core applications.

Chapter 13: Security in ASP.NET Core - This chapter addresses best practices for ensuring backend security, focusing on authorization management, the use of HTTPS and SSL/TLS certificates, and protection against common vulnerabilities in web applications.

Chapter 14: Securing Web Applications Effectively - This chapter examines concepts of authentication and authorization, data protection, and encryption. It includes techniques to prevent attacks such as XSS and CSRF, along with logging and monitoring strategies to enhance security.

Chapter 15: Error Handling - The chapter addresses error management, focusing on logging and tracking techniques, exception handling, and the use of monitoring and diagnostic tools to ensure stable and reliable applications.

Chapter 16: Containerization for Seamless Deployment - This chapter introduces containerization, focusing on Docker integration with ASP.NET Core. It explores container orchestration with Kubernetes and managing configurations and dependencies for effective deployments.

Chapter 17: Building Responsive User Interfaces with Blazor - This chapter provides an overview of frontend development with Blazor. It explains its usage modes, such as server, WebAssembly, and hybrid, highlighting the differences to make informed decisions during software design. It also delves into techniques for building modern and responsive user interfaces.

Chapter 18: Advanced User Interfaces with Blazor - This chapter is a natural continuation of the previous one, as it delves into concepts such as form validation, parameter passing between pages, state management, and JavaScript interoperability, providing a comprehensive overview of frequently used functionalities.

Chapter 19: Debugging, Testing, and Performance Tuning - This chapter provides an overview of debugging and testing strategies, focusing on best practices for identifying and resolving issues. It includes performance optimization techniques to improve application responsiveness and efficiency.

Code Bundle and Coloured Images

Please follow the link to download the *Code Bundle* and the *Coloured Images* of the book:

<https://rebrand.ly/a43ec5>

The code bundle for the book is also hosted on GitHub at <https://github.com/bpbpublications/Architecting-ASP.NET-Core-Applications>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at <https://github.com/bpbpublications>. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in

touch with us at :

business@bpbonline.com for more details.

At www.bpbonline.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at business@bpbonline.com with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit www.bpbonline.com. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit www.bpbonline.com.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



Table of Contents

1. Introduction to ASP.NET Core

Introduction

Structure

Objectives

Overview of ASP.NET Core

Setup our PC for ASP.NET Core

A new project ASP.NET Core

ASP.NET Core Web App

Modify the project

ASP.NET Core Web App

Blazor Web App

Codebase and complexity

Technical debt

Recognize the complexity

Causes of complexity

Rules to avoid complexity

Software design principles

SOLID principles

Single responsibility principle

Open/Closed Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

Don't Repeat Yourself

Keep It Simple, Stupid

Conclusion

Points to remember

Multiple choice questions

Answer key

Key terms

2. Basics of ASP.NET Core

Introduction

Structure

Objectives

Structure of project

Server project

Client project

Pages folder

wwwroot folder

Location of Blazor JS scripts

Middleware and request pipeline

Recommended middlewares

Terminal middleware

Built-in middleware

Application of basic design patterns

Command Pattern

Component-based architecture

TaskManagerApp component

TaskModel class

Event-driven programming

Define an event argument

Define a class to manage events

Wire up event handlers

Trigger events

Option Pattern

Conclusion

Points to remember

Multiple choice questions

Answer key

Key terms

3. Architectures and Core Components

Introduction

Structure

Objectives

Overview of different architectures

Monolithic architecture

Microservices architecture

Serverless architecture

Event-driven architecture

Design principle-based architectures

Understanding Model-View-Controller

MVC pattern

Implementation

Understanding Clean Architecture

When Clean Architecture fits and fails

Clean Architecture in practice with ASP.NET Core project

Summarizing up Clean Architecture

Understanding Domain-Driven Design

Ubiquitous Language

Entities and value objects

Aggregates

Domain events

Summing up DDD

Understanding Hexagonal Architecture

Project structure

Implementing Hexagonal Architecture

Conclusion

Points to remember

Multiple choice questions

Answer key

Key terms

4. Designing RESTful APIs

Introduction

Structure

Objectives

Concepts of RESTful APIs

REST and RESTful

Understanding REST Principles

Statelessness

Client-Server architecture

Uniform interface

HTTP methods and status codes

HTTP methods

Status codes

ASP.NET Core Web API

Endpoints and controllers in ASP.NET Core

ASP.NET Core controller-based API

Controller and ControllerBase classes

Attributes

Binding source parameter inference

Problem details to return error status codes

ASP.NET Core minimal API

Minimal APIs and MVC in ASP.NET Core

Minimal APIs

Minimal API with MVC

Model

Controller

View

Minimal API structure

Best practices in designing RESTful APIs

Client side

Services

Server side

Extension method refactoring

Conclusion

Points to remember

Multiple choice questions

Answer key

Key terms

5. Implementing Routing in ASP.NET Core

Introduction

Structure

Objectives

Introduction to routing in ASP.NET Core

Understanding routing

Understanding endpoints

Routing in minimal APIs

URL mapping to controllers and actions

Route parameters

Attribute routing

Configuring and customizing routing

Defining route templates

Catch-all route template

Route constraints and parameters

Route file-path template

Constraint resolver

Applying route constraints

Attribute routing with templates

Parameter transformers

Best practices and tips for routing

Implementation of versioning

Keeping routes simple and predictable

Using attribute routing sparingly

Leveraging route constraints wisely

Thoughtfully organizing controllers and actions

Implementing RESTful routing
Handle missing routes gracefully

Conclusion

Points to remember

Multiple choice questions

Answer key

Key terms

6. Middleware and Extensibility

Introduction

Structure

Objectives

Dependency injection concepts

Inversion of Control

Service lifetimes

Implementation of dependency injection in ASP.NET Core

Configuring dependency injection in ASP.NET Core

Understanding middleware components

Early termination of the request pipeline

RUN delegates

Building a middleware pipeline with web application

Middleware ordering

Middleware components in Program.cs

Common middleware scenarios

Development environment

Staging environment

Production environment

Creating custom middleware

Define the middleware class

Create an extension method for the middleware

Register the middleware in the pipeline

Short circuit the pipeline

Extending the dependency injection

Understanding the need for extension

Strategies for Extending DI

Custom service factories

Multiple instances of the same interface

The options pattern

Best practices for extending dependency injection

Conclusion

Points to remember

Multiple choice questions

Answer key

Key terms

7. Architectural Principles

Introduction

Structure

Objectives

Architectural principles and anti-patterns

Antipattern

Big ball of mud

God Object

Golden Hammer

Lava flow

Architectural patterns

Architectural principles

Modularity

Scalability

Resilience

Building a software system

Definition of requirements

Requirements analysis

Decisions and trade-offs

Making decisions by ADRs

Purpose and content

- Structure of ADR*
- Benefits of ADR*
- Usage of ADR*
- System design*
- Implementation of system design*
- Testing of software*
- Deployment of software*
- Post-deployment care*
 - Software maintenance*
 - Software gardening*
- Definition of termination criteria*
- Backend architecture layering
 - The essence of layering*
 - Layering the backend architecture*
 - Clean Architecture as separated applications*
 - Benefits of layered architecture*
- Functionality oriented techniques
 - Vertical slicing*
 - Feature vertical slicing*
- Connection between SOLID principles and architecture
- Conclusion
- Points to remember
- Multiple choice questions
 - Answer key*
- Key terms

8. GoF Design Patterns

- Introduction
- Structure
- Objectives
- Application of GoF design patterns in ASP.NET Core
- Singleton Pattern in ASP.NET Core application
 - Concepts*

How it works

Implementation of Singleton Pattern

Entity Framework Core overview

ORM

Key features of Entity Framework Core

Using Entity Framework Core

The data model

Create the Database Context

Factory Method Pattern with EF Core

Concepts

How it works

Concrete problem

Implementation

Define the Factory interface

Create concrete classes

Create SQLite-specific Factory class

Create in-memory-specific Factory class

Configure dependency injection

Usage in application

Decorator Pattern for logging

Concepts

How it works

Concrete problem

Implementation

Observer Pattern for application state

Concepts

How it works

Concrete problem

Implementation

Strategy Pattern for managing payments

Concepts

How it works

Concrete problem

Implementation

Conclusion

Points to remember

Multiple choice questions

Answer key

Key terms

9. CQRS in Architecture

Introduction

Structure

Objectives

Introduction to CQRS

Command and query separation

Event sourcing integration

Implementing CQRS in practice

Product management

Project structure

Web API

Domain models

Application

Infrastructures

Adding the event sourcing

Event store

Modify command handlers

Projections

Visualize results

Integrate with backend services

Build the UI components

Implementing CQRS in practice: Summary

Conclusion

Points to remember

Multiple choice questions

Answer key

Key terms

10. Modular Monolith

Introduction

Structure

Objectives

Module design

Key characteristics

Project structure

Detailed project structure

Principles of modular design

Defining boundaries and responsibilities

Communication between modules

Method calls

Service broker

Guidelines

Shortcuts

Define clear interfaces

Encapsulate module functionality

Code reviews

Dependency management

Configuration and startup

Testing

Logging and monitoring

Deployment and updates

Conclusion

Multiple choice questions

Answer key

Points to remember

11. SignalR in Real-time Web Applications

Introduction

Structure

Objectives

SignalR

Communication server-client

Named pipes

Message queues

Shared memory

Server-sent events

Long polling

WebSockets

SignalR overview

SignalR communication

Example use cases for SignalR

SignalR in .NET9

Basic use of SignalR

Setting up the SignalR Hub

Client side SignalR

Configuring SignalR at startup

Messages clients

Single client communication

Group client communication

Message pack transmission

Build a basic example

Add the UI logic

Securing SignalR application

MessagePack security

Server configuration

Client configuration

Custom resolver

Conclusion

Points to remember

Multiple choice questions

Answer key

Key terms

12. Automated Testing

Introduction

Structure

Objectives

Importance of testing

Types of tests

V-Model

Verification

Requirement design

System design

Architecture design

Module design

Coding

Validation

Black-box testing

Advantages

Limitations

Implementation

Conducting effective tests in ASP.NET Core

Unit testing

Arrange-Act-Assert

Behaviour-Driven Development

Integration testing

End-to-end testing

Acceptance testing

With SpecFlow

Load testing

Interpreting results

Conclusion

Points to remember

Multiple choice questions

Answer key

Key terms

13. Security in ASP.NET Core

Introduction

Structure

Objectives

Best practices for backend security

Confidentiality, integrity and availability

The three Au's

Securing configuration and secrets

Protecting configuration files

Encrypting sensitive data

Logging and monitoring

Implementing Serilog logging

Monitoring

Use middleware

Application Insights

Authorization management

Implementing authentication

Implementing authorization

Using JSON Web Token

Usage of HTTPS and SSL/TLS certificates

Protecting against common attacks

Cross-site scripting

SQL injection

Points to remember

Conclusion

Multiple choice questions

Answer key

Key terms

14. Securing Web Applications Effectively

Introduction

Structure

Objectives

Authentication and authorization

Authentication using JWT token

JWT token

Use of JWT

Using JWT in ASP.NET Core authentication

Securing WebAPI

Client side usage

Data protection and encryption

Principles and rules of encryption

Encryption in ASP.NET Core

Configuring and using data protection

Use the data protection system

Implementing encryption with AES

Cross-site scripting and cross-site request forgery protection

Cross-site scripting vulnerability

Mitigating XSS vulnerability

Cross-site request forgery vulnerability

Mitigating CSRF vulnerability

Conclusion

Points to remember

Multiple choice questions

Answer key

Key terms

15. Error Handling

Introduction

Structure

Objectives

Logging and error tracking

Error Notification System

Serilog with notification sinks

Best practices

Post-mortem analysis

- Preparation and data collection with Serilog*
- Incident description*
- Root cause analysis*
- Immediate corrective actions*
- Future improvements*
- Documentation and sharing*
- Post-mortem report*
- Review and feedback*
- Exception handling
 - Error versus exceptions*
 - Exceptions*
 - Centralized error handling with middleware*
 - Custom error pages*
 - Exception filters*
- Monitoring and diagnostic tools
- Conclusion
- Points to remember
- Multiple choice questions
 - Answer key*
- Key terms

16. Containerization for Seamless Deployment

- Introduction
- Structure
- Objectives
- Introduction to containerization
 - Microservices architecture*
 - Key benefits of containerization*
 - Steps to achieve containerization*
 - Install Docker Desktop*
- Docker and ASP.NET Core integration
 - The Dockerfiles*
 - Create a containerized website*

Containerizing Web APIs

Orchestrating containers with Kubernetes

Deployment files

Services files

How it works

How to use

Handling configuration and dependencies

Conclusion

Points to remember

Multiple choice questions

Answer key

Key terms

17. Building Responsive User Interfaces with Blazor

Introduction

Structure

Objectives

What is Blazor

Universal compatibility and accessibility

Simplified updates and maintenance

Remote access and collaboration

A web-based solution for .NET

The Razor syntax rules

Blazor Server

Architecture and communication model

Lifecycle and state management

Performance and scalability

Development experience

Blazor Webassembly

What is WebAssembly

Structure of WebAssembly

Validation in WebAssembly

Execution in WebAssembly

- Binary format*
- Text format*
- Architecture of Blazor WebAssembly*
- Razor component*
- Performance and optimization*
- Blazor Hybrid
 - Blazor Hybrid with WPF*
 - Blazor Hybrid with Windows Forms*
 - Blazor Hybrid with MAUI*
- Blazor render modes
 - Blazor InteractiveServer*
 - Blazor InteractiveWebAssembly*
 - Blazor InteractiveAuto*
 - Blazor StreamRendering attribute*
 - Blazor United*
- Conclusion
- Points to remember
- Multiple choice questions
 - Answer key*
- Key terms

18. Advanced User Interfaces with Blazor

- Introduction
- Structure
- Objectives
- Input form validation
 - Form validation*
 - Data annotations*
 - Implementing validation in Blazor*
 - Custom validation*
- Passing parameters through application pages
 - Parameters*
 - Pages parameters*

- Components parameters*
- Cascading parameters*
- EventCallback < T >*
- JavaScript interop
 - Calling JavaScript from Blazor*
 - Calling Blazor function from JavaScript*
 - Passing complex data*
 - Asynchronous operations*
 - Advanced use of JavaScript interop*
- Basic state management
 - Store data in memory*
 - Create the state management service*
 - Use the service in a Blazor components*
 - Store data server side*
 - Store data in browser storage*
 - Local and session storage*
- Conclusion
- Points to remember
- Multiple choice questions
 - Answer key*
- Key terms

19. Debugging, Testing, and Performance Tuning

- Introduction
- Structure
- Objectives
- Introduction to debugging
 - Debugging in ASP.NET Core*
 - Debugging tools*
 - Debugging techniques*
- Testing strategies
 - Importance of testing in ASP.NET Core*
 - Unit testing*

Integration testing

End-to-end testing

Performance testing

Continuous testing

Debugging best practices

Performance tuning

Identifying performance bottlenecks

Memory leak discovery techniques

Blazor Webassembly memory management

Algorithm optimization

Optimizing application code

Daily tuning

Conclusion

Points to remember

Multiple choice questions

Answer key

Key terms

Index

CHAPTER 1

Introduction to ASP.NET Core

Introduction

In this chapter, we will understand how to configure our PC for ASP.NET Core application development, while also addressing a pervasive yet often overlooked issue: complexity. We will explore how to identify and mitigate this problem, which frequently infiltrates software projects. We will also examine the importance of adhering to best practices and implementing rules to steer clear of code complexity. By doing so, we aim to maintain a clean, comprehensible, and sustainable codebase, ultimately highlighting why such practices are imperative.

Structure

In this chapter, we will cover the following topics:

- Overview of ASP.NET Core
- Codebase and complexity
- Software design principles

Objectives

The objective of this chapter is to provide a thorough exploration of ASP.NET Core, a powerful and versatile framework for building web applications. We will begin by the installation of Visual Studio with their customizations and options.

Moreover, we will examine the relationship between codebase and complexity. As projects evolve and grow in scale, managing codebase complexity becomes increasingly paramount. We will explore strategies for mitigating complexity, including technical debts, recognize complexity, and some rules to avoid complexity.

Furthermore, we will emphasize the application of key software engineering principles such as SOLID, and KISS. By dissecting these principles within the context of ASP.NET Core development, we can see how they serve as guiding philosophies for writing clean, modular, and efficient code. This chapter aims to equip readers with the knowledge and skills necessary to start developments in ASP.NET Core.

Overview of ASP.NET Core

ASP.NET Core is a cross-platform, high-performance, open-source framework for building modern, cloud-enabled, Internet-connected apps.

With ASP.NET Core, you can build web apps and services, **Internet of Things (IoT)** apps, and mobile backends. As a cross-platform framework, the application can run on Windows or Linux, servers or PCs, and even Raspberry. We can also choose to deploy to the cloud or on-premises and run on a .NET environment, which is a warranty of success.

There are two main parts to an ASP.NET Core application:

- The backend with the logic, WebAPI, database connections, and so on.
- The frontend with the graphic part of the application. This is the browser's part.

However, to be able to run all these things, we have to install .NET, an environment that allows us to build and manage our code and

some other things. Let us see how to set up our PC.

Setup our PC for ASP.NET Core

The easiest way to start building our ASP.NET Core application is to download **Visual Studio (VS)** and install it.

At the moment of writing this book, the website to download VS is:

<https://visualstudio.microsoft.com/downloads>

We will download an executable and run it, as shown in the following figure:

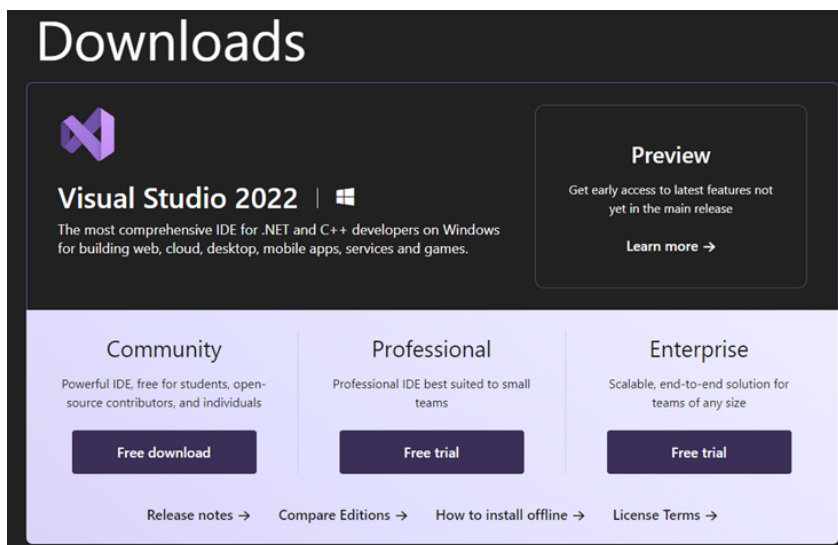


Figure 1.1: Download screen

For our purposes, the Community is enough, but for the examples in this book We will use the Professional or Preview. The differences are really minimal. In practice, the Community version is a Professional version but can only be used for non-paid projects.

Alternatively, you can also install VS Code. VS Code is not an **integrated development environment (IDE)** like VS, but it is almost the same and with a little bit of effort you can use it freely. After downloading the installation file and running it, this screen will appear after a while:

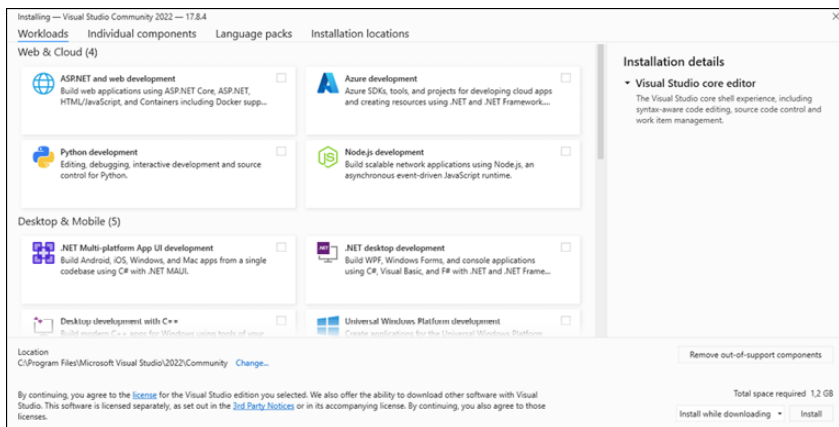


Figure 1.2: Installation choices

As you can see, you have to choose some options according to Done your development choices. In our case, we need to check the box in the square of *ASP.NET and web development*. You can run the Visual Studio Installer application again to add new functions.

The installation will start after pressing the button **Install** in the lower-right part of the application, then a screen with a progress bar will appear. It shows the installation progress of VS environment as you can see in the following figure:

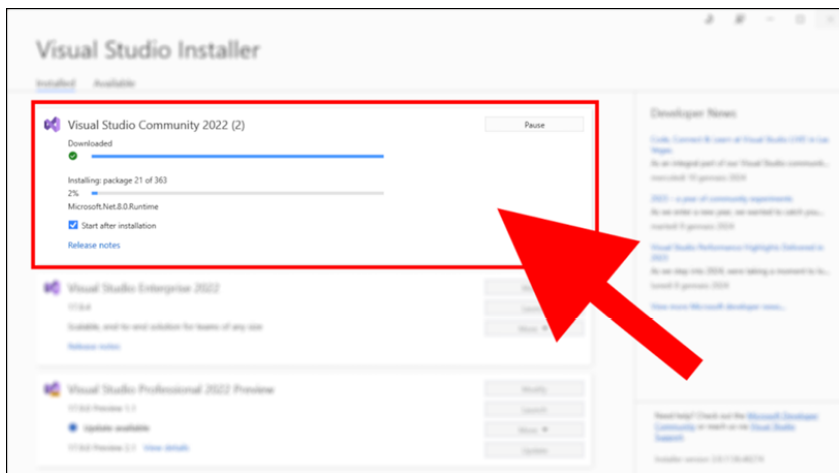


Figure 1.3: Installation progress

When the VS installation is complete, the screen you see will be automatically displayed as we can see in [Figure 1.4](#):

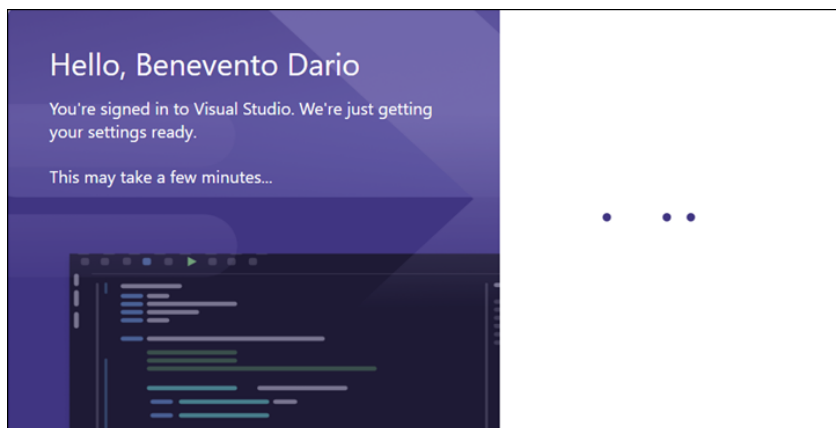


Figure 1.4: Starting Visual Studio

After a little configuration (you are already logged with your Microsoft account or it will ask you to login with a Microsoft account), the starting screen will appear as shown in the following figure:

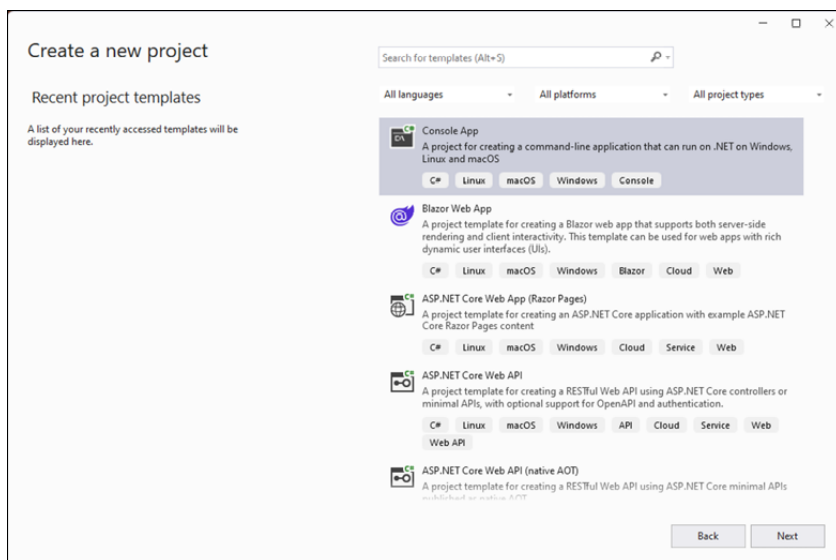


Figure 1.5: New project

A new project ASP.NET Core

At this point, all we have to do is choose a project template, which will allow us to have a correctly configured project skeleton to

begin development.

To have an ASP.NET Core project, we have three possibilities, as follows:

- Choose ASP.NET Core Web App (Model-View-Controller) template
- Choose ASP.NET Core Web App (Razor Pages) template
- Choose Blazor Web App template.

The first uses the **Model-View-Controller (MVC)** model template that uses the MVC design pattern. In short, the pattern uses the concept of **separation of concerns (SoC)**, asserting that the software should be separated based on the work it performs.

The second uses the Razor pages as the front end, which makes coding page-focused scenarios easier and more productive than using classic MVC with controllers and views.

The third is the new concept of frontend that supports server-side rendering, client interactivity and WebAssembly.

ASP.NET Core Web App

We will review these three patterns in more depth in [Chapter 3, Architecture and Core Components](#). For the moment, we will start with the first template. Choose ASP.NET Core Web App (Model-View-Controller).

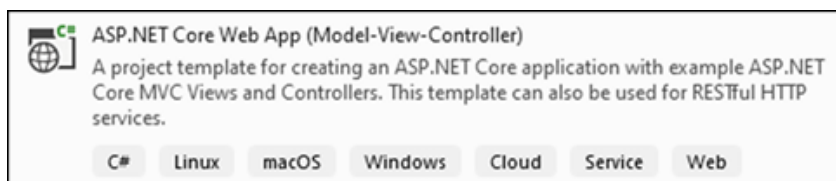


Figure 1.6: Project template

The process to create a new solution in VS consists of two fundamental steps. In [Figure 1.7](#), we can give the name to our first application (for instance **MyFirstProject**) as follows:

Configure your new project

ASP.NET Core Web App (Model-View-Controller) C# Linux macOS Windows Cloud Service Web

Project name
MyFirstProject

Location
C:\Users\drago\Source\Repos

Solution name
MyFirstProject

☐ Place solution and project in the same directory

Project will be created in "C:\Users\drago\Source\Repos\MyFirstProject\MyFirstProject\"

Back Next

Figure 1.7: New project first step

Click on **Next** button, and the VS wizard will ask us some further information, as follows:

Additional information

ASP.NET Core Web App (Model-View-Controller) C# Linux macOS Windows Cloud Service Web

Framework
.NET 9.0 (Standard Term Support)

Authentication type
None

☒ Configure for HTTPS

☐ Enable container support

Container OS
Linux

Container build type
Dockerfile

☐ Do not use top-level statements

☐ Enlist in .NET Aspire orchestration

Back Create

Figure 1.8: New project last step

At this point, click on **Create** button with the default values to start Visual Studio, as follows:

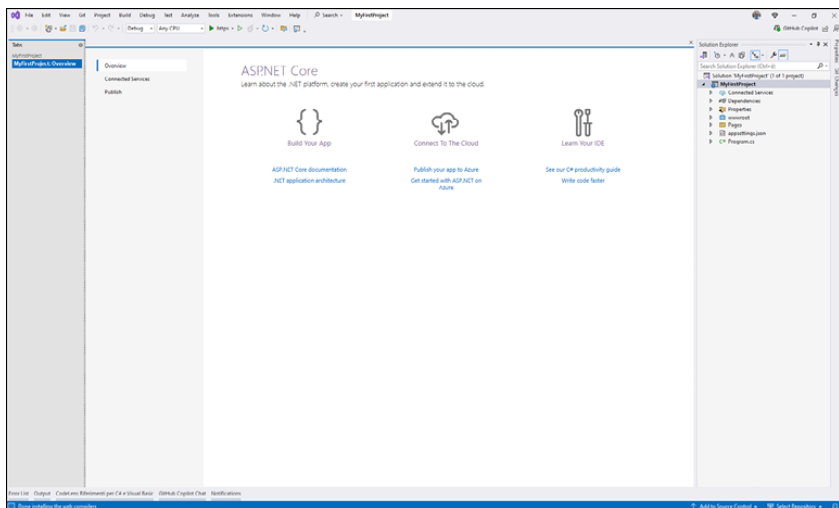


Figure 1.9: New project on Visual Studio

At this point, we can launch the program by pressing *F5* on the keyboard or pressing the button in the upper part of the screen in Visual Studio represented by a solid green triangle. There are two green triangles: one solid-colored and one outlined. The solid-colored one is the one we are interested in, the other is for starting without the debugging but it is not our case in this part of the book. The output will be as follows:

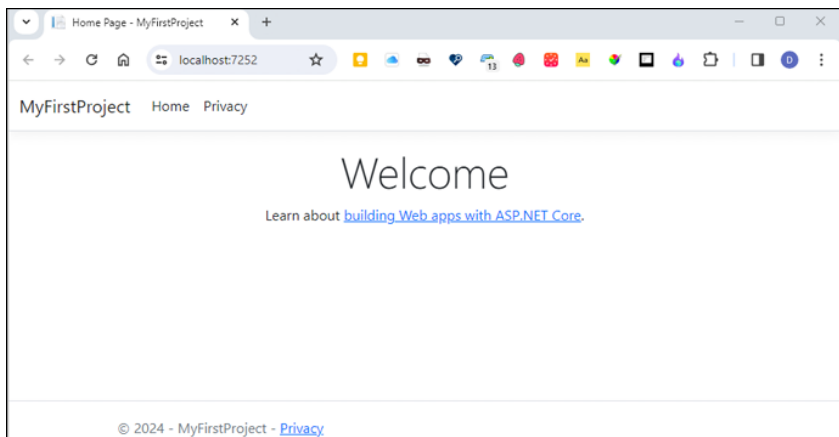


Figure 1.10: New project run

Modify the project

At this point, we can customize the project. In Visual Studio we have on the right side the Solution Explorer where we can see the tree of files and folders. When creating a new ASP.NET Core project using the standard template, a default project structure is generated following the MVC pattern. We will focus on the Views/Home folder where we will find two particular files, as follows:

- Index.cshtml
- Privacy.cshtml

In the context of an ASP.NET Core project, the **Index.cshtml** file is often associated with the default home page and is generally tied to the **Index** action of a controller. One of its key concepts is the use of the MVC pattern, which separates the application logic into three distinct components, as follows:

- **Model:** Represents data and business logic.
- **View:** Handles presentation and user interaction.
- **Controller:** Coordinates user requests and manages application flow.

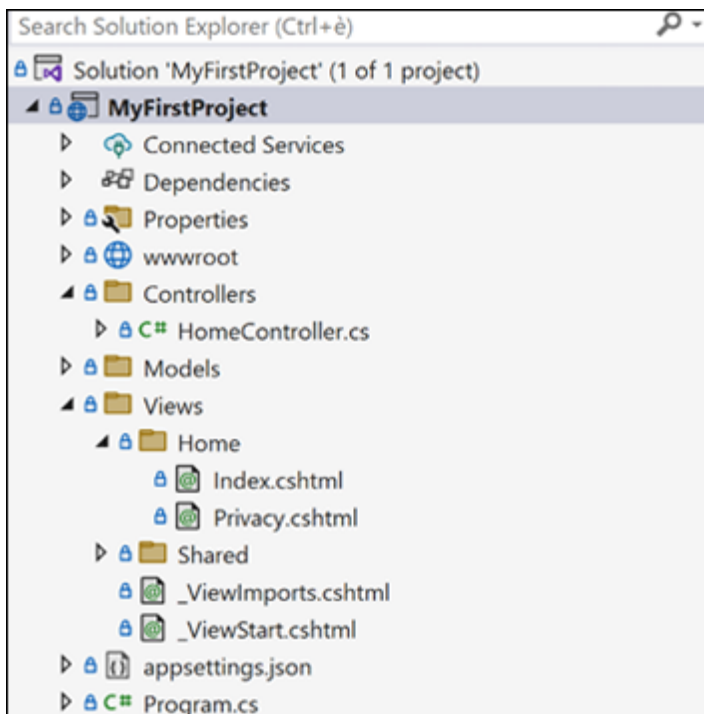


Figure 1.11: Project tree in Visual Studio 2022

We will discuss this pattern in detail in [Chapter 3, Architecture and Core Components](#). For now, it is enough to understand that each page defined by a template file with **cshtml** extension will be powered by an action (i.e. a particular method) in a controller. In our case, we can find an **Index** action in **HomeController.cs** inside the **Controllers** folder. If we open the file **Index.cshtml**, the file content will be as follows:

1. @{
2. ViewData["Title"] = "Home Page";
3. }
4. <div class="text-center">
5. <h1 class="display-4"> Welcome </h1>
6. <p> Learn about building Web apps with ASP.NET Core . </p>
7. </div>
- 8.

In this piece of code, we can easily recognize the HTML parts. The razor syntax is in the first three lines, where you can see the @ and curly brackets. In this defined area, we can insert C# code. We can modify the page as follows:

```
1. @{
2.     ViewData["Title"] = "Home Page";
3.     var MyTitle = "Architecting ASP.NET Core applications";
4. }
5. <div class="text-center">
6.     <h1 class="display-4">@MyTitle</h1>
7.     <p>Learn about <a href="https://learn.microsoft.com/aspnet/core">building Web apps with ASP.NET Core</a>.</p>
8. </div>
9.
```

As you can see, a C# variable **MyTitle** is defined in the C# area on top and used in **h1** tag with the **@MyTitle** directive.

ASP.NET Core Web App

The project templates ASP.NET Core Web App (Razor Pages) to create a similar structure but more compact. We can encounter only a Pages folder in the project tree, as we can see in the [Figure 1.11](#) just down in this paragraph. In this template, a **cs** file will be linked to a **cshtml** page to allow C# code in a separate file. The rest is almost identical to the previous one, as follows:

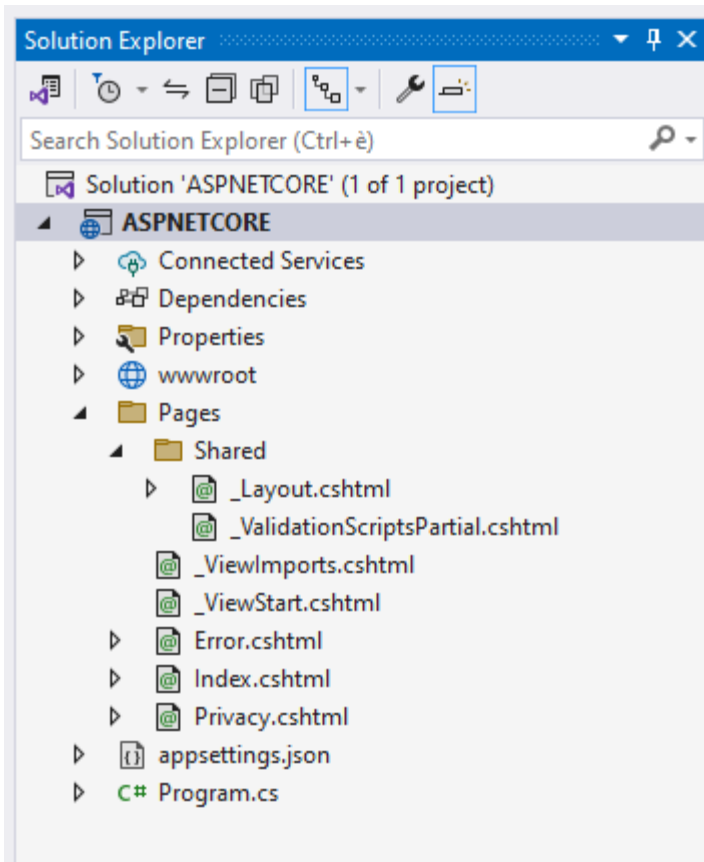


Figure 1.12: ASP.NET Core template project tree

Blazor Web App

The Blazor WebApp template is fundamentally different from the previous ones. Blazor is the most innovative front-end language provided by ASP.NET Core environment. In this case too, we can mix C# code with HTML. We will see this language, its structures, and how to use it in [Chapter 17, Building Responsive User Interfaces with Blazor](#), and [Chapter 18, Advanced User Interfaces with Blazor](#). For the purpose of this book, we will use this template in the particular way of Blazor WebAssembly, because it (in .NET9) will create two separate projects.

There are some settings to provide before create the project. In the initial screen, we can see that starting the wizard of the project template should be as follows:

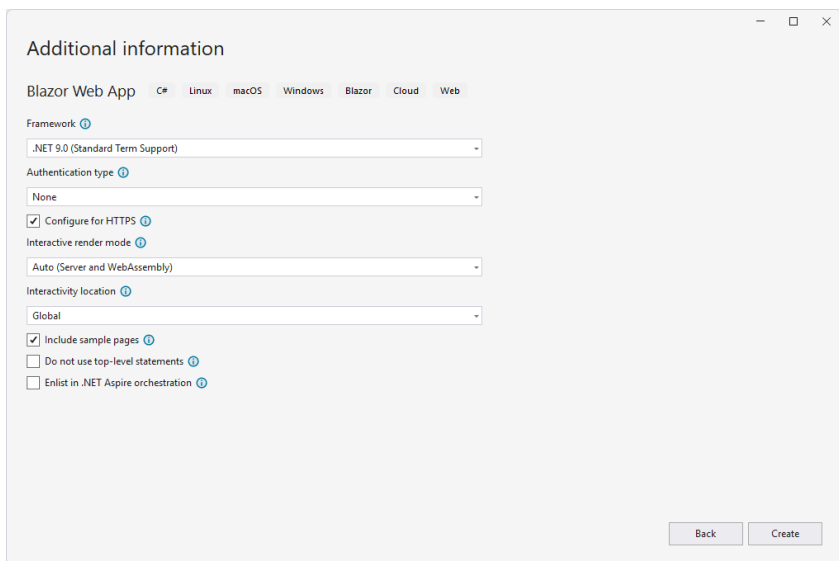


Figure 1.13: Blazor new project

The solution is shown as follows:

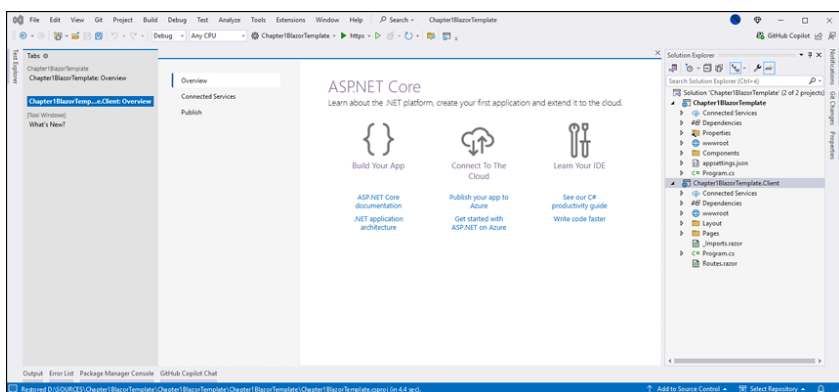


Figure 1.14: Blazor project in Visual Studio 2022

In [Chapter 2, Basics of ASP.NET Core](#), we will start to analyze this template because we need to customize it for our next project.

Codebase and complexity

Software complexity refers to the various factors in a system that make it tricky to understand and change. This can include things like tangled connections between parts, confusing paths through the

code, a lot of code to sift through, and how different parts of the system interact. It can also be made worse by things like poorly explained code, inconsistent writing styles, and using outdated technology. As software gets bigger and more complicated, managing this complexity becomes really important to keep things running smoothly and make it easier for everyone working on the software to understand and modify it.

One of the multiple definitions of complexity is as follows:

Software complexity is anything related to the structure of a system that makes it hard to understand and modify the system.

— John Ousterhout

We can find this phrase in the book *Philosophy of Software Design* which covers the entire panorama of how to properly design software.

From a software developer's perspective, the complexity of a codebase refers to the intricacy and difficulty in understanding, maintaining, and extending the software. It encompasses various factors as follows:

- Size of the codebase
- Architecture
- Interdependencies between different components
- Quality of documentation
- Presence of technical debt
- Clarity of code
- Level of abstraction

High complexity can hinder productivity, introduce bugs, and make it challenging for developers to collaborate effectively. Therefore, developers often strive to manage and reduce complexity through practices like modularization, refactoring, and adherence to coding standards.

Technical debt

We want to draw special attention to the technical debt. **Technical debt** refers to the accumulation of compromises made during the

coding process, often arising from decisions to take shortcuts, add dependencies recklessly and without paying particular attention, or ignore elements deemed unnecessary in the moment, with an attitude of indifference towards potential consequences. It represents the deferred effort required to address these shortcuts and deficiencies at a later stage, which may ultimately impede the efficiency, maintainability, and scalability of the codebase. This metaphorical debt, akin to a financial burden, accrues interest over time as the complexities and repercussions of these initial compromises become more apparent. Addressing technical debt necessitates investing resources to refactor, optimize, and enhance the codebase, thereby mitigating risks and ensuring the long-term viability and sustainability of the software project.

Underestimating the technical debt of a codebase is highly consequential because the devastating effects of this debt compound exponentially. Technical debt accumulates interest over time, much like financial debt, resulting in increased complexity, decreased maintainability, and heightened risk of project failure. Ignoring or downplaying technical debt may lead to significant challenges in the future as the deferred effort required to address it grows larger and more difficult to manage. Thus, acknowledging and proactively managing technical debt is crucial for ensuring the long-term health and success of a software project. In any cases, there are tools that help developers manage and measure technical debt more effectively, keeping it under control.

Recognize the complexity

There are different types of complexity in code, such as cyclomatic complexity and perceived complexity. Cyclomatic complexity is defined as being proportional to the number of decision points, like IF statements and FOR loops. Perceived complexity is actually the one we are most concerned with. When we start a new codebase, we do not know anything about the whole code. The first thing that will be asked us to do, will be a little modification in the codebase. There are three main symptoms of complexity in a codebase, as follows:

- The first symptom of complexity is when we need to modify code in multiple areas for what appeared to be a simple

adjustment.

- The second symptom of complexity entails the **cognitive load** on developers, indicating the depth of understanding required to complete a task. Complexity cannot always be measured by lines of code; sometimes, a method requiring more code can actually simplify the task by reducing cognitive load.
- The last symptom of complexity is the ambiguity regarding which code segments must be altered for task completion, or what information developers need for success. **Unknown unknowns** refer to information gaps that are difficult to identify or resolve. A fundamental objective of effective design is to ensure the system's workings are intuitively understandable.

Causes of complexity

There are two causes of complexity: **obscurity** and **dependencies**.

Obscurity arises when crucial information remains hidden or unclear. Obscurity frequently coincides with dependencies, where the presence of a dependency is not readily apparent. Inconsistency exacerbates obscurity, particularly when the same variable name serves multiple purposes, making developers uncertain about its intended use. The necessity for extensive documentation often signals inadequacies in the design. Simplifying the system design is the most effective strategy for mitigating obscurity.

A dependency arises when a specific segment of code cannot undergo comprehension and modification in isolation. It interconnects with other code components, necessitating their consideration and/or modification if alterations are made to the given code. Dependencies constitute an integral aspect of software and cannot be entirely eradicated. Therefore, a primary objective of software design entails minimizing the quantity of dependencies and ensuring that those which persist are straightforward and apparent in nature.

Rules to avoid complexity

Complexity does not stem from a solitary catastrophic error;

instead, it accumulates through numerous small increments. It emerges as a result of the gradual buildup of hundreds or thousands of minor dependencies and obscurities over time. The incremental nature of complexity poses challenges in its management and control.

Avoiding complexity in code involves adopting various strategies and best practices throughout the software development lifecycle. Following are some key approaches:

- **SOLID principles:** Following SOLID principles helps us to keep code clean and well organized.
- **Don't Repeat Yourself (DRY) principle:** Eliminate code duplication by abstracting common functionality into reusable components, functions, or libraries. DRY code is easier to maintain and less prone to bugs.
- **Keep It Simple, Stupid (KISS) principle:** Strive for simplicity in design and implementation. Avoid over-engineering and unnecessary complexity by choosing the simplest solution that meets the requirements.
- **Clear and consistent naming:** A simple rule will be very powerful. It is said that one of the biggest challenges for a software developer is to correctly name the parts of the code you are writing. Use descriptive and consistent naming conventions for classes, methods, variables, and other elements. Clear naming reduces ambiguity and improves code readability.
- **SoC:** Divide your codebase into distinct modules or layers, each responsible for a specific functionality. SoC promotes maintainability and facilitates changes without impacting other parts of the system.
- **Continuous refactoring:** Regularly refactor code to improve clarity, remove duplication, and reduce complexity. Refactoring should be integral to the development process to maintain code quality and readability.
- **Design patterns:** Break down your code into modular components following design patterns provided by the **Gang of Four (GoF)**. GoF refers to the authors of the book *Design*

Patterns: Elements of Reusable Object-Oriented Software) or the most common ones. Using well known design patterns, reduces obscurity because the structure of the pattern is well known. Design patterns help organize code, reduce complexity, and promote maintainability.

- **Architectural patterns:** Choose appropriate architectural patterns such as layered architecture, MVC, Clean Architecture, or **Domain-Driven Design (DDD)** architecture based on the requirements of your application. Architectural patterns provide guidelines for structuring and organizing code, reducing complexity, and promoting scalability.

By incorporating these principles and practices into your software development process, you can effectively avoid complexity in code and create maintainable, scalable, and robust applications.

Software design principles

In this section, we will make an excursus on the most used principles in software. We will explore the SOLID principles: KISS and DRY.

SOLID principles

In software development, SOLID principles are considered the basis for developing correct, robust, flexible, and maintainable systems. Remember that in a software codebase, **system** refers to the overall structure that includes all the parts working together. It includes what users see, like buttons and features, as well as the behind-the-scenes stuff like libraries and databases. Essentially, it is everything that makes the software work, organized, and connected.

They are a set of five design principles in object-oriented programming and each principle focuses on a specific aspect of software design, providing guidelines for writing cleaner and more robust code. The principles are discussed in the next section.

Single responsibility principle

The **single responsibility principle (SRP)** suggests that a class should have only one reason to change. This means that each class should be responsible for only one part of the system's behavior,

and if something were to change in that part of the behavior, the class would be the only one to be modified.

For example, if you have a class that handles business logic and at the same time manages the user interface, this would violate SRP. This is because if the business logic changes, you would be forced to modify the part that handles the user interface as well, making the class responsible for more than one aspect.

In ASP.NET Core, this translates to controllers, services, models and middleware having clear and specific responsibilities and well separated. For instance, a controller should handle only interaction with the model and presentation of data, while a service should focus on business logic.

In the following example, we can see a simple console application with some classes managing only one aspect of the whole process:

```
1. using System;
2. using System.Collections.Generic;
3.
4. Product coffee = new Product("Coffee", 3.50m);
5. Product croissant = new Product("Croissant", 2.00m);
6. Product muffin = new Product("Blueberry Muffin", 2.50m);
7.
8.     InventoryManager     inventoryManager     = new
    InventoryManager();
9. inventoryManager.AddProduct(coffee);
10. inventoryManager.AddProduct(croissant);
11. inventoryManager.AddProduct(muffin);
12.
13. SalesRegister salesRegister = new SalesRegister();
14.     salesRegister.ProcessOrder(new List<Product> { coffee,
    croissant, muffin });
15.
16. inventoryManager.RemoveProduct(croissant);
17.
18.
19. public record Product (string Name, decimal Price);
20.
```

```

21. public class InventoryManager
22. {
23.     public void AddProduct(Product product)
24.     {
25.         // Logic to add the product to the store inventory
26.         Console.WriteLine($"Added product: {product.Name}, Price:
           {product.Price}");
27.     }
28.
29.     public void RemoveProduct(Product product)
30.     {
31.         // Logic to remove product from store inventory
32.         Console.WriteLine($"Removed product: {product.Name}");
33.     }
34. }
35.
36. public class SalesRegister
37. {
38.     public void ProcessOrder(List<Product> products)
39.     {
40.         decimal totalPrice = 0;
41.         foreach (var product in products)
42.         {
43.             totalPrice += product.Price;
44.             Console.WriteLine($"Sold product:
           {product.Name}, Price: {product.Price}");
45.         }
46.         Console.WriteLine($"Total: {totalPrice}");
47.     }
48. }

```

In this example, we manage a coffee shop inventory. Each class as **InventoryManager** and **SalesRegister**, manages only one aspect of the coffee shop respecting the principle.

Open/Closed Principle

The **Open/Closed Principle (OCP)** advocates that a class should be

open for extension but closed for modification. This means that the behavior of a software entity should be extendable without modifying its source code. In practical terms, this principle suggests designing software modules in a way that allows new functionalities to be added through inheritance, composition, or other means, without altering the existing codebase. This promotes software systems that are resilient to changes and easier to maintain over time.

For example, instead of modifying existing code to accommodate new features, the OCP encourages creating new classes that inherit from existing ones or implementing new interfaces. This way, the original code remains untouched, reducing the risk of introducing bugs and maintaining the system's stability.

ASP.NET Core supports this principle through middleware and filters that allow the application's behavior to be extended without modifying existing code. The use of services and dependency injection promotes code extensibility.

The following code example extends the previous code about coffee shop inventory changing the pricing strategy without touching the existing code:

```
1. using System;
2. using System.Collections.Generic;
3.
4. Product coffee = new Product("Coffee", 3.50m);
5. Product croissant = new Product("Croissant", 2.00m);
6. Product muffin = new Product("Blueberry Muffin", 2.50m);
7.
8. List<Product> order = new List<Product> { coffee, croissant,
   muffin };
9.
10. SalesRegister salesRegister = new SalesRegister(new
    StandardPricingStrategy());
11. salesRegister.ProcessOrder(order);
12.
13. // Changing the pricing strategy to discounted pricing
14. salesRegister = new SalesRegister(new
    DiscountedPricingStrategy(10)); // 10% discount
```



```
15. salesRegister.ProcessOrder(order);
16.
17. public record Product (string Name, decimal Price);
18.
19. // Interface class for pricing strategies
20. public interface PricingStrategy
21. {
22.     decimal CalculatePrice(List<Product> products);
23. }
24.
25. // Concrete implementation of pricing strategy
26. public class StandardPricingStrategy : PricingStrategy
27. {
28.     public decimal CalculatePrice(List<Product> products)
29.     {
30.         decimal totalPrice = 0;
31.         foreach (var product in products)
32.         {
33.             totalPrice += product.Price;
34.         }
35.         return totalPrice;
36.     }
37. }
38.
39. // New pricing strategy for discount on total price
40. public class DiscountedPricingStrategy : PricingStrategy
41. {
42.     private decimal discountPercentage;
43.
44.     public DiscountedPricingStrategy(decimal
discountPercentage)
45.     {
46.         this.discountPercentage = discountPercentage;
47.     }
48.
49.     public decimal CalculatePrice(List<Product> products)
50.     {
51.         decimal totalPrice = 0;
```

```

52.     foreach (var product in products)
53.     {
54.         totalPrice += product.Price;
55.     }
56.     decimal discount = totalPrice * (discountPercentage /
100);
57.     return totalPrice - discount;
58. }
59. }
60.
61. public class SalesRegister
62. {
63.     private PricingStrategy pricingStrategy;
64.
65.     public SalesRegister(PricingStrategy pricingStrategy)
66.     {
67.         this.pricingStrategy = pricingStrategy;
68.     }
69.
70.     public void ProcessOrder(List<Product> products)
71.     {
72.         decimal totalPrice =
pricingStrategy.CalculatePrice(products);
73.
74.         Console.WriteLine("Receipt:");
75.         foreach (var product in products)
76.         {
77.             Console.WriteLine($"- {product.Name}:
${product.Price}");
78.         }
79.         Console.WriteLine($"Total: ${totalPrice}");
80.     }
81. }

```

In this example, we have introduced the **PricingStrategy** abstract class, which defines a method **CalculatePrice**. We then have two concrete implementations of this abstract class: **StandardPricingStrategy** and **DiscountedPricingStrategy**. This allows us to calculate the price of the products differently based on

the selected strategy.

Now, when we want to introduce a new pricing strategy, we can create a new class that inherits from **PricingStrategy** and implement the **CalculatePrice** method accordingly, without modifying existing code. This adheres to the OCP, as the **SalesRegister** class is closed for modification but open for extension through new pricing strategies.

Liskov Substitution Principle

The **Liskov Substitution Principle (LSP)** requires that objects of a derived class seamlessly substitute objects of the base class. In other words, derived classes should substitute their base classes without altering the desirable properties of the program.

Following are the key points of the LSP:

- **Behavioral subtyping:** Subtypes must exhibit behavior that is consistent with the behavior of the supertype. This means that methods should behave in a way that is expected by the client code when the subtype replaces the supertype.
- **Inheritance:** The LSP deals with the relationship between classes in an inheritance hierarchy. It emphasizes that inheritance should model an *is-a* relationship, where a subclass is a specialized version of its superclass. This ensures that the behavior of the superclass is preserved in its subclasses.
- **Contract:** Subclasses must uphold the contract established by the superclass. This includes obeying preconditions, postconditions, and invariants defined by the superclass. Clients of the superclass should be able to rely on these contracts when working with subclasses.
- **Avoidance of ad hoc modifications:** The principle discourages making **ad hoc modifications** to a subclass to fix issues that arise when substituting it for its superclass. Instead, the design should be reconsidered to ensure that all subclasses conform to the same contract as the superclass.

In ASP.NET Core, this can be applied in inheritance situations, such as when extending the functionality of a base controller with one

specific to a particular area of the application.

Refer to the following code:

```
1. using System;
2. using System.Collections.Generic;
3.
4. Product coffee = new Drink("Coffee", 3.50m);
5. Product croissant = new Pastry("Croissant", 2.00m);
6. Product muffin = new Pastry("Blueberry Muffin", 2.50m);
7.
8. List<Product> order = new List<Product> { coffee, croissant,
    muffin };
9.
10. SalesRegister salesRegister = new SalesRegister();
11. salesRegister.PrintProductDescriptions(order);
12.
13.
14. // Abstract class representing a product
15. public abstract class Product
16. {
17.     public string Name { get; }
18.     public decimal Price { get; }
19.
20.     public Product(string name, decimal price)
21.     {
22.         Name = name;
23.         Price = price;
24.     }
25.
26.     public abstract string GetDescription();
27. }
28.
29. // Concrete class representing a drink product
30. public class Drink : Product
31. {
32.     public Drink(string name, decimal price) : base(name, price)
```

```

33. {
34. }
35.
36. public override string GetDescription()
37. {
38.     return $"Drink: {Name}, Price: {Price:C}";
39. }
40. }
41.
42. // Concrete class representing a pastry product
43. public class Pastry : Product
44. {
45.     public Pastry(string name, decimal price) : base(name, price)
46.     { }
47.     public override string GetDescription()
48.     {
49.         return $"Pastry: {Name}, Price: {Price:C}";
50.     }
51. }
52. // SalesRegister class that accepts a list of products and prints
their descriptions
53. public class SalesRegister
54. {
55.     public void PrintProductDescriptions(List<Product>
products)
56.     {
57.         foreach (var product in products)
58.         {
59.             Console.WriteLine(product.GetDescription());
60.         }
61.     }
62. }

```

In this example, we have an abstract class **Product** representing the superclass, and two concrete subclasses **Drink** and **Pastry**. Each subclass provides its implementation of the **GetDescription** method.

We can see that instances of **Drink** and **Pastry** can be used interchangeably with the superclass **Product** within the **PrintProductDescriptions** method of the **SalesRegister** class, demonstrating compliance with the LSP. This means that we can extend the functionality of the **SalesRegister** class by adding new types of products without modifying their existing behavior.

Interface Segregation Principle

The **Interface Segregation Principle (ISP)** advises having specific interfaces for each class.

It emphasizes that classes should not be forced to depend on interfaces they do not use. In simpler terms, it suggests that interfaces should be specific to the needs of the classes that use them rather than overly broad or encompassing multiple functionalities.

In short, we can summarize the ISP as follows:

- **Class-specific interfaces:** Interfaces should be tailored to the specific requirements of class. This means that each class should have access to an interface that provides only the methods they need and nothing more.
- **Avoidance of fat interfaces:** A **fat** interface contains more methods than necessary for a particular class. Such interfaces violate the ISP because classes are forced to depend on methods they do not use, leading to unnecessary coupling and potential complications during maintenance. Furthermore, the classes implementing that fat interface will potentially have more than one responsibility. The principle that would be violated in this scenario is the Single Responsibility Principle, wherein the ISP principle is connected to a dual mandate.
- **Prevention of interface pollution:** Interface pollution occurs when an interface becomes bloated with irrelevant methods to some classes. The ISP advises against this practice and encourages breaking down large interfaces into smaller, more focused ones.

In the following example, we get the last example of the coffee shop, and we extend the concept to ISP:

```

1. using System;
2. using System.Collections.Generic;
3.
4. Product coffee = new Product("Coffee", 3.50m);
5. Product croissant = new Product("Croissant", 2.00m);
6. Product muffin = new Product("Blueberry Muffin", 2.50m);
7.
8. List<ISellable> order = new List<ISellable> { coffee,
   croissant, muffin };
9.
10. SalesEmployee salesEmployee = new SalesEmployee();
11. salesEmployee.ProcessOrder(order);
12.
13.     InventoryManager     inventoryManager     = new
   InventoryManager();
14. inventoryManager.AddInventoryItem(coffee);
15. inventoryManager.RemoveInventoryItem(croissant);
16.
17.
18. // Interface for products that can be sold
19. public interface ISellable
20. {
21.     string GetName();
22.     decimal GetPrice();
23. }
24.
25. // Interface for products that can be managed in inventory
26. public interface IInventoryItem
27. {
28.     void AddToInventory();
29.     void RemoveFromInventory();
30. }
31.
32. // Interface for employees who can perform sales tasks
33. public interface ISalesEmployee
34. {
35.     void ProcessOrder(List< ISellable > products);
36. }

```

```
37.
38. // Interface for employees who can manage inventory
39. public interface IInventoryManager
40. {
41.     void AddInventoryItem(IInventoryItem item);
42.     void RemoveInventoryItem(IInventoryItem item);
43. }
44.
45. // Class representing a product
46. public class Product : ISellable, IInventoryItem
47. {
48.     private string name;
49.     private decimal price;
50.
51.     public Product(string name, decimal price)
52.     {
53.         this.name = name;
54.         this.price = price;
55.     }
56.
57.     public string GetName()
58.     {
59.         return name;
60.     }
61.
62.     public decimal GetPrice()
63.     {
64.         return price;
65.     }
66.
67.     public void AddToInventory()
68.     {
69.         Console.WriteLine($"Added {name} to inventory.");
70.     }
71.
72.     public void RemoveFromInventory()
73.     {
74.         Console.WriteLine($"Removed {name} from inventory.");
```



```

75.     }
76. }
77.
78. // Sales employee class
79. public class SalesEmployee : ISalesEmployee
80. {
81.     public void ProcessOrder(List<ISellable> products)
82.     {
83.         decimal total = 0;
84.         foreach (var product in products)
85.         {
86.             total += product.GetPrice();
87.             Console.WriteLine($"Sold: {product.GetName()} - Price:
{product.GetPrice():C}");
88.         }
89.         Console.WriteLine($"Total: {total:C}");
90.     }
91. }
92.
93. // Inventory manager class
94. public class InventoryManager : InventoryManager
95. {
96.     public void AddInventoryItem(IInventoryItem item)
97.     {
98.         item.AddToInventory();
99.     }
100.
101.     public void RemoveInventoryItem(IInventoryItem item)
102.     {
103.         item.RemoveFromInventory();
104.     }
105. }

```

In this example, we have introduced four interfaces: **ISellable**, **IInventoryItem**, **ISalesEmployee**, and **IInventoryManager**. Each interface represents a specific aspect of the coffee shop system, as follows:

- **ISellable**: Interface for products that can be sold.
- **IInventoryItem**: Interface for products that can be managed

in inventory.

- **ISalesEmployee:** Interface for employees who can perform sales tasks.
- **InventoryManager:** Interface for employees who can manage inventory.

Classes such as **Product**, **SalesEmployee**, and **InventoryManager** implement these interfaces based on their respective functionalities. This ensures that each class adheres to the ISP by depending only on interfaces relevant to its responsibilities.

Dependency Inversion Principle

The **Dependency Inversion Principle (DIP)** states that high-level classes should not depend on low-level classes, but both should depend on abstractions.

Following are the key points of the DIP:

- **High-level modules and low-level modules:** In traditional software design, high-level modules for example, business logic, application workflow often depend on low-level modules for example, database access, external services directly. However, according to DIP, this direct dependency leads to a rigid and fragile design, making it difficult to change and maintain the codebase.
- **Dependency on abstractions:** Instead of depending directly on concrete implementations, high-level and low-level modules should depend on abstractions for example, interfaces, abstract classes. Abstractions define a contract without specifying the details of how the functionality is implemented. This allows for flexibility and decoupling in the system.
- **Abstractions should not depend on details:** Abstractions should not be coupled to the specific implementation details. They should be stable and not affected by changes in the underlying implementation. This ensures the overall design remains resilient to changes and promotes code reuse.
- **Details should depend on abstractions:** Concrete implementations (details) should depend on abstractions

rather than the other way around. This allows different implementations to be easily swapped without affecting the higher-level modules that depend on them.

- **Inversion of Control (IoC):** The *Dependency Inversion Principle* is closely related to the concept of IoC, where the control of flow is inverted from the traditional approach. Instead of high-level modules controlling the flow of execution and instantiating low-level modules, control is delegated to a separate component (e.g., IoC container) that manages the creation and wiring of objects based on their dependencies. In ASP.NET Core, there is a built-in system of dependency injection (we will see this system in [Chapter 6, Middleware and Extensibility](#)) which act as an IoC Container.

In the following code example, we can find an example of DIP linked to a coffee shop:

```
1. using System;
2. using System.Collections.Generic;
3.
4. Product coffee = new Product("Coffee", 3.50m);
5. Product croissant = new Product("Croissant", 2.00m);
6. Product muffin = new Product("Blueberry Muffin", 2.50m);
7.
8. List<Product> order = new List<Product> { coffee, croissant,
   muffin };
9.
10.     IInventoryManager    inventoryManager    = new
   InventoryManager();
11.     IOrderProcessor      orderProcessor      = new
   InventoryService(inventoryManager);
12. SalesService salesService = new SalesService(orderProcessor);
13. salesService.ProcessOrder(order);
14.
15. // Interface representing a service for processing orders
16. public interface IOrderProcessor
17. {
18.     void ProcessOrder(List<Product> products);
19. }
```

```
20.
21. // Interface representing a service for managing inventory
22. public interface IInventoryManager
23. {
24.     void AddToInventory(Product product);
25.     void RemoveFromInventory(Product product);
26. }
27.
28. // Class representing a product
29. public class Product(string Name, decimal Price);
30.
31. // High-level module that depends on IOrderProcessor abstraction
32. public class SalesService
33. {
34.     private readonly IOrderProcessor orderProcessor;
35.
36.     public SalesService(IOrderProcessor orderProcessor)
37.     {
38.         this.orderProcessor = orderProcessor;
39.     }
40.
41.     public void ProcessOrder(List<Product> products)
42.     {
43.         orderProcessor.ProcessOrder(products);
44.     }
45. }
46.
47. // Low-level module that depends on IInventoryManager abstraction
48. public class InventoryService : IOrderProcessor
49. {
50.     private readonly IInventoryManager inventoryManager;
51.
52.     public InventoryService(IInventoryManager inventoryManager)
53.     {
54.         this.inventoryManager = inventoryManager;
```

```

55.     }
56.
57.     public void ProcessOrder(List<Product> products)
58.     {
59.         foreach (var product in products)
60.         {
61.             inventoryManager.RemoveFromInventory(product);
62.             Console.WriteLine($"Sold:{product.Name}-Price:
{product.Price:C}");
63.         }
64.     }
65. }
66.
67. // Concrete implementation of IInventoryManager
68. public class InventoryManager : IInventoryManager
69. {
70.     public void AddToInventory(Product product)
71.     {
72.         Console.WriteLine($"Added {product.Name} to
inventory.");
73.     }
74.
75.     public void RemoveFromInventory(Product product)
76.     {
77.         Console.WriteLine($"Removed {product.Name} from
inventory.");
78.     }
79. }

```

In this example, we have the following:

- **IOrderProcessor** interface representing a service for processing orders.
- **IInventoryManager** interface representing a service for managing inventory.

The **SalesService** class represents a high-level module that depends on the **IOrderProcessor** abstraction, while the **InventoryService** class represents a low-level module that depends on the **IInventoryManager** abstraction.

By depending on abstractions (**IOrderProcessor** and **IInventoryManager**), both high-level and low-level modules are decoupled from concrete implementations, adhering to the DIP. This allows for flexibility and easier maintenance, as different implementations of **IOrderProcessor** and **IInventoryManager** can be swapped without affecting the **SalesService** and **InventoryService** classes.

In summary, integrating SOLID principles into our development brings numerous benefits. By separating concerns, embracing extensibility, relying on abstractions rather than concrete implementations, ensuring module cohesion, and practicing dependency inversion, code becomes more maintainable, extendable, and testable.

In ASP.NET Code, SOLID principles should be the hard core of your software. Do not forget about these principles as they will help you write your code better.

Don't Repeat Yourself

Don't Repeat Yourself (DRY) represents one of the fundamental pillars in software development. Its essence lies in the idea of petition within code by promoting the creation of reusable components and eliminating duplicated logic or data. This is like drying your code.

At the heart of the DRY principle is the importance of maintaining a single source of truth for each piece of information or functionality within a software system. This means developers should avoid duplicating code or information in multiple places.

By adhering to the DRY principle, you can obtain several advantages, as follows:

- **Reduction of redundancy:** Code duplication leads to maintenance challenges and increases the probability of errors. By eliminating redundancy, developers can make their code more manageable and easier to maintain.
- **Improve code reusability:** By abstracting common functionality into reusable components, you can create modular code that can be easily reused across different parts of the application.

- **Maintainability of code:** Having a single source of truth for each functionality simplifies the process of making changes or updates. When you have a single class/function only need to modify the code in one place, reducing the risk of inconsistencies and ensuring that changes are applied uniformly throughout the codebase.
- **Better readability and understandability:** Your code will be enhanced with DRY. Your code tends to be more concise and easier to understand, as it avoids clutter and unnecessary repetition. This makes it easier for developers to reason about the code and reduces cognitive load (see paragraph about complexity).

Overall, the DRY principle plays a crucial role in promoting code quality and maintainability in software development projects. By following this principle, developers can create codebases that are easier to understand, modify, and extend over time.

Take a look at the following example:

```

1. using System;
2.
3. double lengthRectangle = 5;
4. double widthRectangle = 3;
5. double sideSquare = 4;
6.
7. double areaRectangle = CalculateArea(lengthRectangle,
   widthRectangle);
8. double areaSquare = CalculateArea(sideSquare, sideSquare);
9.
10. Console.WriteLine($"Area of rectangle: {areaRectangle}");
11. Console.WriteLine($"Area of square: {areaSquare}");
12.
13. // Function to calculate the area
14. public static double CalculateArea(double length, double width)
15.     => length * width;
```

In this example, we have one single function **CalculateArea** responsible for calculating the area of a rectangle and a square.

Instead of duplicating the logic to calculate the area in multiple

places, we have created one single reusable function.

This adheres to the DRY principle because the logic for calculating the area is not repeated, and we have a single source of truth for this functionality. Furthermore, if there is a bug in the function, we can modify once and for all.

We could improve the function to calculate the area of the square by passing only one of the two arguments and making the second optional. For example, we can choose to put the width parameter as optional with a default value, as follows:

```
1. // Function to calculate the area
2. public static double CalculateArea(
3.     double length, double width = double.MinValue)
4. {
5.     if(width == double.MinValue) width = length;
6.     return length * width;
7. }
```

In this way your code will be understandable and maintainable. An algorithm can be implemented in some different ways. In this case we can choose to assign the width parameter with the same value of length if no argument was passed. On the other hand, we find ourselves subjecting other people who will read our code to a greater cognitive load, having to know exactly and in detail what the function does inside it but this is one of the subjects of *Recognize the complexity* early in this chapter.

Keep It Simple, Stupid

The **Keep It Simple, Stupid (KISS)** principle is a fundamental design principle that emphasizes simplicity and straightforwardness in software development. The essence of the KISS principle is to favor simplicity over complexity whenever possible, aiming to keep systems, designs, and solutions as simple and easy to understand as possible.

The key points of the KISS principle are as follows:

- **Simplicity:** The KISS principle advocates for simplicity in design and implementation. This means avoiding unnecessary complexity, over-engineering, or adding features that are not essential to achieving the desired

outcome.

- **Clarity:** Simple solutions are easier to understand, maintain, and debug. By keeping designs and implementations straightforward, developers can reduce cognitive load and make it easier for themselves and others to comprehend and work with the code.
- **Minimalism:** The principle encourages minimizing the number of dependencies, and components in a system. This helps reduce the risk of errors, improve performance, and enhance overall reliability.
- **Focus on core functionality:** Instead of addressing every possible edge case or scenario, the KISS principle advises focusing on the core functionality and requirements of the system. This allows developers to deliver solutions more efficiently and effectively without getting bogged down by unnecessary complexity.
- **User experience:** Simple designs often result in better user experiences. Users tend to prefer interfaces and systems that are intuitive, easy to navigate, and free from unnecessary clutter or complexity.

As we can see, the KISS principle encourages developers to favor simplicity, clarity, and minimalism in software design and development. More practically, the KISS principle encourages us to keep our methods, classes, or functions small (no more than 30-40 lines), with each method solving only one specific problem.

The dumber our code is, the more intelligence we have to put into *dumbing it down*. Remember that you have to be smart, but the code must remain simple and stupid.

Conclusion

This chapter has provided how to start to write an ASP.NET Core application, a comprehensive overview of software development practices, and delved into its intricacies, advantages, and fundamental concepts. We have explored the importance of understanding codebase complexity and the insidious ways it can manifest, emphasizing the need for developers to be vigilant and

proactive in managing and simplifying their codebase.

Furthermore, we have highlighted the significance of starting off on the right foot by adhering to key principles such as SOLID, KISS, and DRY. These principles serve as guiding lights, helping developers write cleaner, more maintainable code that stands the test of time.

As you begin your journey with architecting an ASP.NET Core application, remember to continually strive for simplicity, clarity, and adherence to best practices.

In the next chapter, we will get to know ASP.NET Core in more depth, starting from the basics. We will explore MVC architecture, the concept of middleware and we will start with some design patterns. These topics will provide us with a solid foundation for developing robust and scalable web applications.

Points to remember

- Understand the setup process on your PC, how to create a new project, and grasp the basics of MVC architecture for web development.
- Be aware of technical debts and how they accumulate over time. Recognize complexity in code and follow established rules to avoid it, ensuring maintainability and scalability.
- Familiarize yourself with essential design principles such as SOLID, DRY, and KISS. Applying these principles leads to cleaner, more maintainable codebases.

Multiple choice questions

1. In an ASP.NET Core Web App (Model-View-Controller) project, what does the Model part represent?
 - a. Represents the business logic and data
 - b. Represents the visual part
 - c. Represents the styling part
 - d. Represents the coordination between user requests and application flow

2. In a very extensive codebase, a symptom of complexity is:

- a. Having code in C#
- b. Having a very small project
- c. Having a very large project
- d. Having an ASP.NET Core project

3. When writing code, what action should we absolutely take?

- a. Copying pieces of code and pasting them where I need them to reuse already written code
- b. Grouping common code into a single function/class to have a single source of truth
- c. Keeping the visual parts of the application separate from each other
- d. None of the above

Answer key

Key terms

- **Visual Studio:** An IDE used for writing, compiling, and debugging code.
- **Codebase complexity:** The difficulty in understanding, maintaining, and modifying an application's code.
- **Technical debts:** Design or implementation choices that may cause issues in the long term, such as unoptimized code or lack of documentation.
- Software engineering principles
- **SOLID:** An acronym representing five object-oriented design principles for writing maintainable and flexible code.
- **DRY:** An acronym advocating for the avoidance of duplicating code, which can lead to maintenance issues and inconsistencies.
- **KISS:** A design principle emphasizing the importance of keeping solutions simple and clear.

CHAPTER 2

Basics of ASP.NET Core

Introduction

In this chapter, we will learn how to start a new application in ASP.NET Core. It is easy to start with the proposed MVC architecture. When the application will scale up, it will be easy to adapt our project to new architectural choices. According to the Blazor Web App template (covered in *Chapter 1, Introduction to ASP.NET Core*) provided from .NET9, we can analyze the base structure. This template has a really good separation of concerns and provides a base for our future project. In this way, we can explore the code more confidently. Furthermore, the Blazor Web App template project is only for the front end and does not manage any kind of API directly in the default template.

Structure

In this chapter, we will cover the following topics:

- Structure of project
- Middleware and request pipeline

- Application of basic design patterns

Objectives

The objectives of this chapter are to understand the fundamental components that constitute the structure of a Blazor WebApp project, providing a comprehensive understanding of its organization. We aim to explore the middleware and request pipeline intricacies within the ASP.NET Core applications, elucidating their roles in handling incoming requests and orchestrating responses. Additionally, this chapter endeavors to elucidate the practical application of basic design patterns within the Blazor framework as Component-based, Event-driven, and Command Patterns, empowering readers to leverage established solutions for common software design challenges. By the end of this chapter, readers should possess a solid grasp of how to structure their projects effectively, optimize middleware configurations, and employ essential design patterns to enhance code maintainability and scalability.

Structure of project

As we said before, we will refer to the Blazor templates. The Blazor Web App project template serves as a unified starting point for leveraging Razor components in constructing diverse styles of web user interfaces, whether they are rendered server-side or client-side. This template merges the strengths of Blazor Server and Blazor WebAssembly hosting models, offering server-side rendering, streaming rendering, improved navigation, form handling, and the flexibility to introduce interactivity through either Blazor Server or Blazor WebAssembly on a component-specific basis.

When **client-side rendering (CSR)** and **interactive server-side rendering (interactive SSR)** are chosen during app creation, the project template employs the Interactive Auto render mode. Initially, it utilizes interactive SSR while downloading the .NET app bundle and runtime to the browser. Subsequently, upon activating the .NET WebAssembly runtime, rendering transitions to CSR.

By default, the Blazor Web App template facilitates static and interactive server-side rendering within a single project. If

Interactive WebAssembly rendering is also enabled, an additional client project (**.Client**) is included for WebAssembly-based components. The output from the client project is downloaded and executed on the client side. Components utilizing the Interactive WebAssembly or Interactive Auto render modes must reside in the **.Client** project.

Server project

The structure of the server project contains some folders. The first one is the **Components** folder, which contains the following two subfolders:

- **Layout folder:** This folder contains layout components and stylesheets as follows:
 - **MainLayout component (MainLayout.razor):** The primary layout component of the application.
 - **MainLayout.razor.css:** Stylesheet for the main layout according to the CSS isolation.
 - **NavMenu component (NavMenu.razor):** Facilitates sidebar navigation, including NavLink component (NavLink) for rendering navigation links to other Razor components. The NavLink component highlights the currently displayed component.
 - **NavMenu.razor.css:** Stylesheet for the navigation menu.
- **Pages folder:** Houses routable server-side Razor components (**.razor**) structured with **@page** directive. It includes the following:
 - **Error component (Error.razor):** Implements the Error page.
 - **Home component (Home.razor):** Implements the Home page.
 - **Weather component (Weather.razor):** Implements the Weather forecast page only, for example.

Furthermore, the Components folder contains the following files:

- **App component (App.razor):** This razor page serves as the

root component of the application, comprising HTML `<head>` markup, the Routes component, and the Blazor `<script>` tag. It is the initial component loaded by the application.

- **Routes component (Routes.razor):** Set up routing via the Router component, intercepting browser navigation for client-side interactive components and rendering the requested page. In this part, we can add our personal routes.
- **Imports.razor:** Contains common Razor directives to include in server-rendered app components (`.razor`), such as `@using` directives for namespaces.
- **Properties folder:** This folder contains only the development environment configuration in the `launchSettings.json` file.

Note: In the `launchSettings.json` file, the `http` profile precedes the `https` profile. This setup ensures that when running the application with the .NET CLI, it operates at an HTTP endpoint by default. This ordering facilitates the transition to adopting HTTPS for Linux and macOS users. To start the application with the .NET CLI without specifying the `-lp https` or `--launch-profile https` option, place the `https` profile above the `http` profile in the file.

- **wwwroot folder:** The last relevant folder is `wwwroot`. This folder houses the Web Root folder for the server project, containing the application's public static assets.

The project contains also the `Program.cs` file in its root. This file serves as the entry point for the server project, setting up the ASP.NET Core web application host and containing startup logic, including service registrations, configuration, logging, and request processing pipeline. This file contains the following:

- Services for Razor components are added through calls to `AddRazorComponents`. `AddInteractiveServerComponents` adds services supporting rendering Interactive Server components, while `AddInteractiveWebAssemblyComponents` adds services supporting rendering Interactive WebAssembly components.
- `MapRazorComponents` discovers available components and defines the application's root component, which is, by default, the App component (`App.razor`).

- **AddInteractiveServerRenderMode** configures interactive SSR, while **AddInteractiveWebAssemblyRenderMode** configures Interactive WebAssembly render mode.
- The root of the server project also contains the App Settings Files **appsettings.Development.json** and **appsettings.json** that contains configuration settings for the server project.

Client project

When using a Blazor WebApp application, the template adds a project suffixed as **.Client** that represents the client-side application that runs in the browser. This project contains the UI components, Razor pages, and logic written in C# that interact with the user. It leverages WebAssembly to run .NET code directly in the browser, offering a rich, interactive experience without needing a full page reload. The **.Client** project typically includes services, routing, and state management, making it the responsible part for the frontend user interface and client-side functionality.

Pages folder

The Client project contains the folder **Page**, which is responsible for containing the client-side Razor components (**.razor**) that are routable within the application. Each page's route is specified using the **@page** directive. By default, the project includes the Counter component (**Counter.razor**), responsible for implementing the Counter page as example.

Note: The component folder structure within the **.Client** project differs from the main project in the Blazor Web App because the main project adheres to the standard ASP.NET Core project structure. As such, the main project must accommodate other assets relevant to ASP.NET Core projects, which may not directly relate to Blazor.

In contrast, the **.Client** project is exclusively focused on Blazor and does not need to integrate as extensively with non-Blazor features and specifications of ASP.NET Core. Consequently, it employs a simpler component folder structure. However, you have the flexibility to organize the component folders in the **.Client** project according to your preferences. You can opt to mimic the folder layout of the main project within the **.Client** project, though keep in

mind that adjustments to namespaces might be necessary for certain assets like layout files if components are relocated to different folders than those prescribed by the project template.

wwwroot folder

The **wwwroot** folder for the client-side project contains the application's public static assets, including app settings files (**appsettings.Development.json**, **appsettings.json**) that furnish configuration settings specific to the client-side project.

Program.cs

As the server project, the Client contains a file named **Program.cs**. This serves as the entry point for the client-side project, establishing the WebAssembly host and encapsulating the project's startup logic, encompassing service registrations, configuration, logging, and request processing pipeline.

_Imports.razor

The last file in this folder is **_Imports.razor**. This file incorporates common Razor directives necessary for WebAssembly-rendered app components (**.razor**), such as **@using** directives for namespaces.

Furthermore, depending on the configuration options chosen, additional files and folders may be present in the application generated from a Blazor Web App project template. For instance, if ASP.NET Core Identity is enabled during app generation, supplementary assets for authentication and authorization features are included.

There are two other project categories of templates based on Blazor. They are Blazor WebAssembly and Blazor Server App. The difference is that those templates are based on .NET7 and the migration to .NET9 should be made by hand and the Blazor WebApp template is .NET9 native. The structure of these three project templates is similar to each other.

Location of Blazor JS scripts

Blazor runs across a JavaScript library. The Blazor scripts are positioned in the main entry point of the single page application. We can see where the scripts are positioned and what their names are, as follows:

- **Blazor Web App template:** In the Blazor Web App template, the Blazor script are located within the **components/App.razor** directory of the project.

```
<script src="_framework/blazor.web.js"> </script>
```

- **Blazor WebAssembly template:** In the Blazor WebAssembly template, the Blazor script resides within the **wwwroot** directory and located in file **wwwroot/index.html**, as shown:

```
<script src="_framework/blazor.webassembly.js"> </script>
```

- **Blazor Server template:** Finally, in the Blazor Server template, the Blazor script is included in the **_Host.cshtml** file, typically located in the **Pages** directory. The script reference is as follows:

```
<script src="_framework/blazor.server.js"> </script>
```

Understanding the locations and roles of these scripts is fundamental for configuring and maintaining Blazor applications across different hosting models.

Middleware and request pipeline

In a Blazor application, middlewares are components or services that intercept and process HTTP requests and responses as they flow through the application pipeline. Blazor is a web framework built on top of ASP.NET Core, which leverages the middleware pipeline provided by ASP.NET Core to handle incoming HTTP requests and outgoing responses.

Middleware components in a Blazor application can perform various tasks, such as authentication, authorization, logging, request/response manipulation, exception handling, and more. They sit between the client and the server-side logic, allowing you to execute code before and after each HTTP request is handled.

Refer to the following figure of middleware call chain:

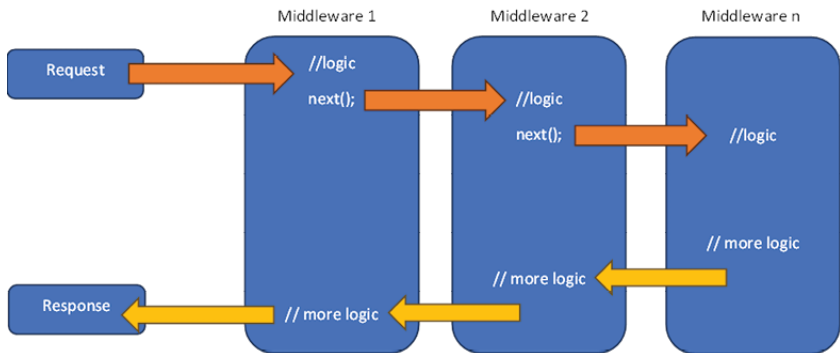


Figure 2.1: *Middleware pipeline*

A simplified explanation of how middleware works in a Blazor application is given as follows:

- **Incoming request:** When a client sends a request to the server, it passes through the middleware pipeline before reaching the Blazor components or pages.
- **Middleware processing:** Each middleware component in the pipeline has the opportunity to inspect and potentially modify the incoming request. This allows tasks, such as authentication or request logging to be performed before the request is passed to the Blazor components.
- **Blazor component execution:** After the request has been processed by the middleware, it is then passed to the appropriate Blazor component or page for handling.
- **Outgoing response:** Once the Blazor component has processed the request and generated a response, it passes back through the middleware pipeline.
- **Middleware processing (Outgoing):** Middleware components also have the opportunity to inspect and modify the outgoing response before it is sent back to the client. This can be useful for tasks, such as adding headers or logging the response.
- **Response sent to client:** Finally, the modified response is sent back to the client.

Middleware in a Blazor application can be configured and added to the pipeline using the **Program.cs** file, similar to how it is done in

ASP.NET Core applications. This allows for a flexible and modular approach to handling HTTP requests and responses, making it easier to add and remove functionality as needed.

For instance, the **UseHttpsRedirection** middleware in ASP.NET Core is used to automatically redirect HTTP requests to their HTTPS counterparts. It is a security measure commonly employed to ensure that sensitive information, such as login credentials or personal data, is transmitted securely over encrypted connections. In [Chapter 6, Middleware and Extensibility](#), we will understand middlewares in more detail.

Recommended middlewares

By default, some middlewares are added to our project ASP.NET Core when we create a new application. The sequence of middleware components added in the **Program.cs** file determines the sequence in which they are invoked for processing incoming requests and, conversely, in reverse order for handling responses. This ordering is vital as it directly impacts the application's security, performance, and functionality. We can add some more middleware as follows:

```
1. /* OMISSIS */
2.
3. app.UseHttpsRedirection();
4. app.UseStaticFiles();
5. app.UseCookiePolicy();
6.
7. app.UseRouting();
8. app.UseRateLimiter();
9. app.UseRequestLocalization();
10. app.UseCors();
11.
12. app.UseAuthentication();
13. app.UseAuthorization();
14.
15. app.Run();
16.
```

When middleware needs to execute before route matching,

UseRouting should be invoked, and the middleware should be positioned prior to the call to **UseRouting**. By using **UseRouting()**, we are setting up the routing middleware to inspect the incoming HTTP request's URL and determine which endpoint should handle it based on predefined route templates. **UseRouting()** configures routing for incoming requests. These route templates typically map URLs to controller action methods. **UseEndpoints** is unnecessary in this scenario, as it is automatically included, as previously described.

Terminal middleware

In ASP.NET Core, terminal middleware plays a critical role in the request processing pipeline. The terminal middleware is the last middleware component in the request processing pipeline. Once terminal middleware is executed, the request processing pipeline completes, and the response is sent back to the client. Its primary role is to handle the final stages of request processing and prepare the response to be sent back to the client. The role of terminal middleware is like a cornerstone. It is responsible for handling the response before it is sent back to the client. This includes finalizing the response headers, content, and any other necessary operations to prepare the response for delivery.

It often includes the error handling logic. If an exception occurs during request processing and is not caught earlier in the pipeline, terminal middleware can catch it and generate an appropriate error response. This ensures that even if an unexpected error occurs, the client receives a meaningful response.

Terminal middleware is also used for cleanup and finalization tasks. This might include releasing resources, logging final request or response details, or performing any other necessary cleanup operations before the response is sent. It is added to the ASP.NET Core application's middleware pipeline using the **UseEndpoints()** method in the **Program.cs**. This method configures the routing middleware to map incoming requests to endpoint handlers, with the terminal middleware being the last component in the pipeline.

When incorporating a terminal middleware, keep in mind the following:

- Ensure the middleware is added after **UseEndpoints**.
- The application must call **UseRouting** (line 1) and **UseEndpoints** (line 3) to ensure proper placement of the terminal middleware, as shown in the following code:

```

1. app.UseRouting();
2. app.MapGet("/", () => "hello world");
3. app.UseEndpoints(e => {});
4. app.Run();

```

Developers can create custom terminal middleware to perform specific tasks during the final stages of request processing. This allows for flexibility in handling responses and performing any necessary cleanup or finalization steps unique to the application’s requirements.

In ASP.NET Core, a **terminal middleware** refers to a middleware component that effectively terminates the middleware pipeline. Once a terminal middleware is executed, no further middleware components are processed in the pipeline. Normally, the last middleware in pipeline is **UseEndpoints**, which handles the routing and dispatching of requests to their corresponding endpoint handlers. If we have a custom terminal middleware, it will be added at the end of the middleware pipeline, then after **UseEndpoints**, to ensure that all preceding middleware has been executed before the request is finalized.

Built-in middleware

ASP.NET Core includes some middleware components. The following table includes a list of built-in middlewares (from the Microsoft website). The Order column offers guidance on where to place the request processing pipeline and specifies conditions under which the middleware might conclude request processing:

Order	Description
1	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
2	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
3	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
4	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
5	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
6	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
7	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
8	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
9	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
10	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
11	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
12	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
13	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
14	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
15	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
16	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
17	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
18	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
19	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
20	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
21	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
22	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
23	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
24	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
25	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
26	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
27	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
28	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
29	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
30	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
31	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
32	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
33	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
34	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
35	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
36	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
37	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
38	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
39	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
40	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
41	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
42	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
43	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
44	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
45	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
46	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
47	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
48	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
49	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
50	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
51	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
52	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
53	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
54	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
55	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
56	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
57	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
58	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
59	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
60	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
61	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
62	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
63	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
64	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
65	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
66	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
67	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
68	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
69	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
70	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
71	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
72	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
73	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
74	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
75	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
76	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
77	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
78	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
79	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
80	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
81	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
82	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
83	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
84	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
85	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
86	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
87	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
88	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
89	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
90	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
91	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
92	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
93	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
94	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
95	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
96	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
97	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
98	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
99	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.
100	ExceptionHandlerMiddleware is deprecated. Terminal for OAuth callbacks.

ASPNETCORE repository. You can find information about the bug at the following address:
<https://github.com/dotnet/aspnetcore/issues/23218>

Before application will log generation information. That is intended states automatically Developer's middle environment as the first middleware in the pipeline when the environment is Development.

Designing supports the logger that provides a default exception page, giving the option to handle pages status code apps, and the default web page for new apps.

Forwarder supports the headers onto the updated fields. Examples: scheme, host, client IP, method.

The ability to request an ASP.NET Core app and its dependencies, such as checking database availability.

Please pass HTTP headers from the incoming request to the outgoing HTTP Client requests.

Ability to change of the middleware pipeline.

Middleware during the request the updated the method.

HTTP Content-Length that not a HTTPS URL.

Default support compact and identify the HTTP clients' parameters in a header. Examples: Forwarded Headers, URL Rewriting.

WebServer if a request with MVC/Razor Pages.

Middleware if the MVC/Razor Pages are fully processed the request.

Default is supporting for that hangs responses basic routing configuration before UseOutputCaching. Use CORS must come before UseOutputCaching.

Response supporting for that hangs responses. This CORS has efficient participation to make it more complex than typically implements for all apps such as Razor Pages because browsers generally set request headers that prevent caching.

Output caching benefits UI apps.

Request support for that on processing frequently.

Response support for that on processing compression.

Request localization support components. Must appear after Routing Middleware when using RouteDataRequestCultureProvider.

Request support for that configuring a request for a single global and per endpoint. UseDeveloperExceptionPage, and UseRouting.

Endpoint for that changing request routes.

Single Page Application (SPA) provide the underlying chain files, MVC and the default page for the SPA.

Session component for that managing sessions.

Static file support for that serving static files and directory browsing.

Default support for that routing based on URL redirecting requests.

Middleware logging of the middleware pipeline Log File Format. (<https://www.w3.org/TR/WD-logfile.html>)

WebSocket WebSockets support to accept WebSocket requests.

Table 2.1: Built-in middlewares from Microsoft official documentation

In [Chapter 6](#), *Middleware and Extensibility*, we will go into detail on

middleware and build one custom.

Application of basic design patterns

Design patterns are essential tools in software development that offer proven solutions to common problems. In web development, understanding and applying these design patterns can significantly enhance the structure, maintainability, and scalability of web applications. In this section, we will explore how several fundamental design patterns are applied within the ASP.NET Core framework.

When we discuss about design patterns, we are referring to the classic design patterns defined by the **Gang of Four**. **Gang of Four (GoF)** refers to the authors of the book, *Design Patterns: Elements of Reusable Object-Oriented Software*. We will discuss two other design patterns used with ASP.NET Core applications and independent of the GoF design patterns (to be discussed in [Chapter 8](#), *GoF Design Patterns*) and one GoF design pattern widely used in ASP.NET Core: the Command Pattern.

Command Pattern

The Command Pattern is the first pattern provided by the GoF (refer to [Chapter 8](#), *GoF Design Patterns* for more details) that we see in this book. In that book, it is defined as behavioral design pattern that allows encapsulating a request as an object, thereby allowing parameterizing clients with queues, requests, and operations. It promotes loose coupling between sender and receiver by separating the object that invokes the operation from the one that performs it. In ASP.NET Core, we can use the Command Pattern to decouple the requester of an operation from the object that performs the action, which can be particularly useful in scenarios, such as handling user requests or implementing undo/execute functionality.

Here is a basic explanation of how you can implement the Command Pattern in ASP.NET Core. Follow the given steps:

1. **Define the command interface:** Start by defining an interface that declares a method for executing the command. This interface could include methods like **Execute()** and **Undo()** if you want to support undo functionality.


```
1. public interface ICommand
2. {
3.     void Execute();
4.     void Undo();
5. }
```

2. **Implement concrete commands:** Implement concrete command classes that implement the **ICommand** interface. Each concrete command represents a specific action or operation that you want to perform.

```
1. public class ConcreteCommand : ICommand
2. {
3.     private readonly CommandBridge bridge;
4.
5.     public ConcreteCommand(CommandBridge bridge)
6.     {
7.         this.bridge = bridge;
8.     }
9.
10.    public void Execute()
11.    {
12.        bridge.PerformExecuteAction();
13.    }
14.    public void Undo()
15.    {
16.        bridge.PerformUndoAction();
17.    }
18. }
```

3. **Create a CommandBridge:** In the above class, we have seen a **CommandBridge** reference but was never mentioned. The **CommandBridge** class contains the actual implementation of the operations that can be performed.

```
1. public class CommandBridge
2. {
3.     public void PerformExecuteAction()
4.     {
```

```

5.     Console.WriteLine("Execute");
6. }
7.
8. public void PerformUndoAction()
9. {
10.     Console.WriteLine("Undo");
11. }
12. }

```

- 4. Dependency injection:** Finally, register your command and receiver classes with the ASP.NET Core dependency injection container so that they can be injected wherever needed.

```

1.     builder.Services.AddSingleton<ICommand,
ConcreteCommand>();
2. builder.Services.AddSingleton<CommandBridge>();

```

We will discuss more about Dependency Injection in [Chapter 6, Middleware and Extensibility](#).

- 5. Client code:** In your ASP.NET Core application, you can instantiate the concrete commands and execute them as needed. You can use a Blazor Razor Page or another service to manage the execution of commands. We can find the page that for simplicity, we can use the **Home.razor** page:

```

1. @page "/"
2. @rendermode InteractiveServer
3. @using CommandEventPattern.Models
4. @inject ICommand command
5. <PageTitle>Home</PageTitle>
6.
7. <h1>Hello, world! </h1>
8. Welcome to your new app.
9.
10. <br/>
11. <button @onclick="onExecute">Execute</button>
12. <button @onclick="onUndo">Undo</button>
13. @code{
14.     private void onExecute()

```

```
15. {
16.     command.Execute();
17. }
18.
19. private void onUndo()
20. {
21.     command.Undo();
22. }
23. }
```

As you can see, we have two buttons at the bottom of the page that launch commands. Those commands are grabbed from the concrete class and relaunched in the bridge class. This bridge class writes something in the console but it can achieve any other action. By following this pattern, you can achieve a flexible and extensible architecture where the requester of an operation is decoupled from the object that performs it. This allows for easier maintenance, testing, and modification of your ASP.NET Core application.

Component-based architecture

Component-based architecture is a fundamental design approach where software systems are built by composing smaller, reusable, and self-contained units called components. Each component encapsulates a specific piece of functionality, such as a user interface element or a logical module, and can be independently developed, tested, and maintained. ASP.NET Core adopts a component-based architecture, where the user interface is divided into reusable and self-contained components. This approach promotes modularity, separation of concerns, and ease of development.

In ASP.NET Core, component-based architecture lies at the heart of the framework's development paradigm. Let us understand how this architecture is applied in ASP.NET Core, as follows:

- **Modularity and reusability:** ASP.NET Core encourages developers to break down the user interface into small and modular components. These components encapsulate the UI elements and the associated logic, promoting code reusability and maintainability. Developers can create a

library of components that can be reused across different parts of the application, leading to more efficient development workflows.

- **Encapsulation and separation of concerns:** Each ASP.NET Core component is self-contained, meaning it manages its own state and behavior without depending heavily on other components. This promotes encapsulation, where the internal workings of a component are hidden from the outside world, reducing complexity and potential conflicts. Furthermore, by separating concerns such as UI presentation, data processing, and event handling, components become easier to understand, test, and maintain.
- **Lifecycle management:** ASP.NET Core provides a well-defined lifecycle for components, allowing developers to hook into various stages of a component's lifecycle, such as initialization, rendering, and disposal. This lifecycle management enables developers to perform tasks, such as data loading, UI updates, and resource cleanup at the appropriate times, ensuring a smooth and responsive user experience.
- **Event-driven interaction:** Components in ASP.NET Core communicate with each other through events and callbacks. By emitting events and handling callbacks, components can react to user actions, communicate changes in state, and coordinate behavior across the application. This event-driven interaction facilitates loose coupling between components, allowing for greater flexibility and extensibility in the application architecture.

The component-based architecture in ASP.NET Core empowers developers to create modular, reusable, and maintainable web applications by breaking down the UI into smaller, self-contained components. These components encapsulate the UI elements and the associated logic, promoting code reuse, separation of concerns, and hierarchical composition. Through well-defined lifecycle management and event-driven interaction, ASP.NET Core Components work together to create dynamic and responsive user experiences.

The purpose of the following example, our component, will be to manage a task list, hiding all the complexity of saving the list, retrieving it from a hypothetical database and displaying it on screen. The example will not take into consideration (for the moment) the backend part but only the frontend componentization. We will see later in the book how to handle this part correctly. At this moment, the example only serves to show how to close the complexity of a certain task in a component.

TaskManagerApp component

For a practical approach, we will use Blazor as the frontend framework. The component will be as follows:

```
1. @rendermode InteractiveServer
2. @using Component.Models
3. <div>
4.     <h1>Task Manager</h1>
5.     <ul>
6.         @foreach (var task in tasks)
7.         {
8.             <li>@task.Name</li>
9.         }
10.    </ul>
11.    <div>
12.        <input type="text" @bind="@newTaskName" />
13.        <button class="btn btn-primary"
14.            @onclick="AddTask">Add Task</button>
15.    </div>
16.
17. @code {
18.     string newTaskName;
19.     List<TaskModel> tasks = new List<TaskModel>();
20.
21.     void AddTask()
22.     {
23.         tasks.Add(new TaskModel { Name = newTaskName });
24.         newTaskName = string.Empty;
25.     }
```

26. }

TaskModel class

A simple model class to represent a task will be as follows:

```
1. public class TaskModel
2. {
3.     public string Name { get; set; }
4. }
5.
```

In this example, we have a **TaskManagerApp** component that serves as the main container of our elements. It contains two main parts, as follows:

- A **TaskList** made with `<ul>` (only for reminder, the `<ul>` is an unordered list in **html**) for displaying task list.
- A form able to add new tasks.

Each part encapsulates specific functionality, promoting modularity and reusability. As the main principle, use this kind of pattern to separate properly components and functionalities. This concept should be extended to the backend too. If we embrace this approach, the whole chain should be involved as follows:

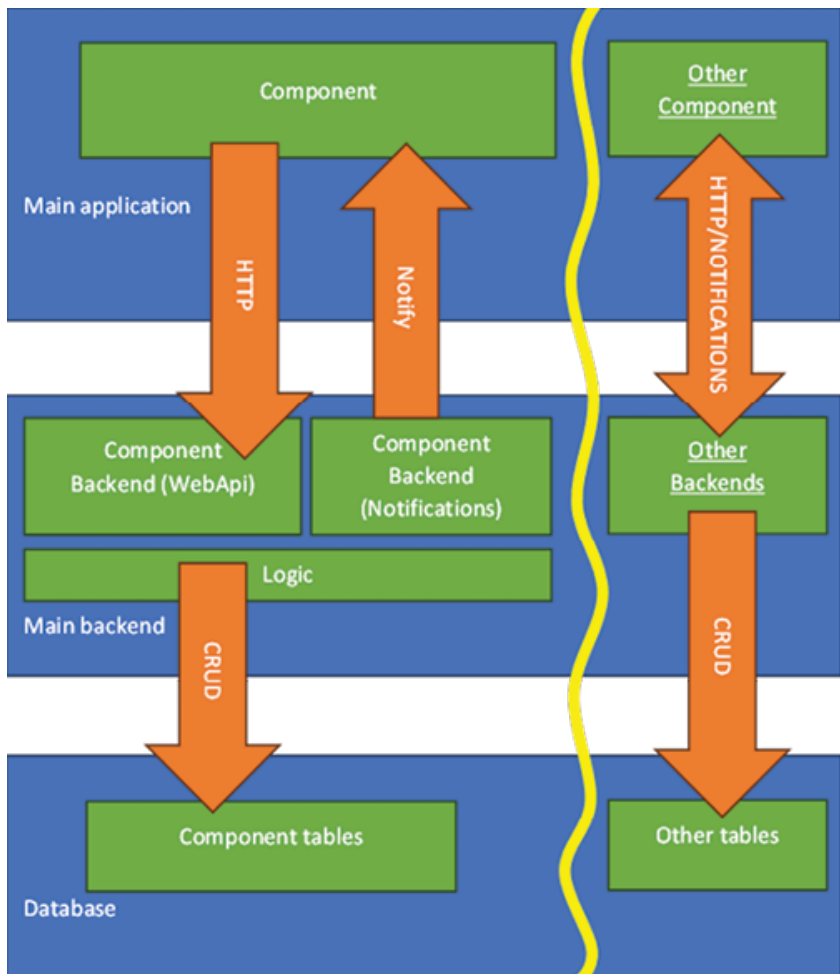


Figure 2.2: Schema of vertical slicing component

This example of blocks talks about components that vertically provide a single functionality (Refer to [Chapter 7, Architectural Principles](#), for more details). In this case, we take operations on a database as an example, but we could have any other type of service or even a physical device.

Event-driven programming

Event-driven programming in ASP.NET Core revolves around handling events that occur during the execution of a web application. These events can be triggered by various actions, such

as incoming HTTP requests, changes in the application state, user actions, and so forth.

Following are some important aspects of event-driven programming in ASP.NET Core:

- **Middleware:** As seen before, ASP.NET Core utilizes a middleware system to handle incoming HTTP requests. Each middleware can be configured to respond to specific events during the request lifecycle, such as the beginning of the request, handling the request, and completing the response. This allows for performing specific operations in response to events like the arrival of a new request.
- **Controller event handling:** In ASP.NET Core, controllers are responsible for handling HTTP requests and often generate events during their lifecycle (we will cover this in [Chapter 4, Designing RESTful APIs](#)). For instance, we can define events that occur before the execution of an action in the controller or after an action has been successfully executed. These events enable performing operations before or after-action execution, such as logging activities or handling exceptions.
- **Application lifecycle events:** ASP.NET Core provides application lifecycle events that allow handling significant events during application startup, execution, and shutdown. These events can be used for tasks such as resource initialization, service registration, and resource cleanup when the application is being shut down.
- **Integration with other event-driven systems:** ASP.NET Core can easily integrate with other event-driven systems, such as message brokers or event streaming services. This enables building reactive applications that dynamically respond to events generated both internally within the application and from external sources.

Overall, event-driven programming in ASP.NET Core provides a powerful way to handle the complexity of modern web applications, allowing for more efficient event management and better system scalability.

In the following sections, we will create a simple example of event-driven programming in a Blazor application where users can

register, and when a user registers successfully, an event will be raised and event handlers will perform actions, such as sending an email notification and logging the registration.

Define an event argument

Create a class to define the event arguments. This class will hold information about the user registration, as follows:

```
1. public class UserRegisteredEventArgs : EventArgs
2. {
3.     public string Username { get; set; }
4.     public string Email { get; set; }
5. }
```

Define a class to manage events

Create a class to manage the events. This class will contain the event and methods to raise the event. This is shown in the following code:

```
1. public class UserEvents
2. {
3.     public event EventHandler<UserRegisteredEventArgs>
        UserRegistered;
4.     public void RegisterUser(string username, string email)
5.     {
6.         // Simulate user registration process
7.         Console.WriteLine($"User'{username}'registered
            email'{email}'");
8.         // Raise the UserRegistered event
9.         OnUserRegistered(new UserRegisteredEventArgs
10.             { Username = username, Email = email });
11.     }
12.     protected virtual void
        OnUserRegistered(UserRegisteredEventArgs e)
13.     {
14.         UserRegistered?.Invoke(this, e);
15.     }
16. }
```

Wire up event handlers

In the **Program.cs** file, wire up the event handlers to the event when configuring services, as shown in the following code:

```
1. builder.Services.AddSingleton < UserEvents > ();
```

Register it as Singleton service because we want to manage events globally across the application.

Trigger events

In a page or component or service where user registration occurs, inject the instance of the **UserEvents** class and call the **RegisterUser** method. For instance, we can link it to a button as follows:

```
1. @using EventDriven.Models;
2. @inject UserEvents UserEvents
3. <PageTitle>Home</PageTitle>
4.
5. <h1>Hello, world!</h1>
6.
7. Welcome to your new app.
8.
9. <br />
10. <h2>User registered: @UserRegistered</h2>
11.
12. <br/>
13. <br/>
14. <br/>
15. <label for="username">User Name</label>
16. <br/>
17. <input id="username" type="text" @bind="@UserName" />
18. <br/>
19. <label for="useremail">User Email</label>
20. <br/>
21. <input id="useremail" type="email" @bind="@UserEmail" />
22. <br/>
23. <br/>
24. <button @onclick="RaiseEvent">Register user</button>
```

For our purpose, we can put everything in the **Home.razor** page with two input text for user email and user password.

Note: If you use Blazor WebApp template, remember to put the appropriate directive `@rendermode` on the top of the page.

As said in the previous section (refer to *Component-based architecture* at point *Modularity and reusability*), we would then create a component able to handle user registration and call the **RegisterUser** method of the **UserEvents** class whenever a new user registers. This is a basic example of event-driven programming in an ASP.NET Core application.

Option Pattern

The Option Pattern in ASP.NET Core is a way to handle application configuration in a flexible and safe manner. This pattern allows you to define and manage configuration options through an options class and access those options within the application.

Following are the steps to summarize how this pattern works:

1. **Defining options:** First, we need to create a class that represents the configuration options. This class should have properties that correspond to the different configuration parameters you want to manage. For example, if you want to manage database connection settings, the class might contain properties like **ConnectionString**, **MaxConnections**, etc.

```
1. public class MyOptions
2. {
3.     public string ConnectionString { get; set; }
4.     public int MaxConnections { get; set; }
5.     // Other configuration properties
6. }
```

2. **Configuring options:** Configuration options are typically defined in the **appsettings.json** file but can be overridden by other configuration providers such as environment variables, command-line parameters, etc.

```
1. {
2.     "MyOptions": {
3.         "ConnectionString": "MyConnectionString",
4.         "MaxConnections": 10
5.     }
6. }
```

```
5. }  
6. }  
7.
```

- 3. Registering options:** In the application startup, you need to register the configuration options with the ASP.NET Core dependency injection system.

The following line is to be added in the **Program.cs** file:

```
1. builder.Services.Configure<MyOptions>(options =>  
2.     builder.Configuration.GetSection("MyOptions").Bind(options));
```

- 4. Using options:** Now, you can inject the configuration options into any class in your project where you need them.

```
1. public class MyClass  
2. {  
3.     private readonly MyOptions _options;  
4.     public MyClass(IOptions<MyOptions> options)  
5.     {  
6.         _options = options.Value;  
7.     }  
8.     public void DoSomething()  
9.     {  
10.        Console.WriteLine($"ConnectionString:  
11.            {_options.ConnectionString}");  
12.        Console.WriteLine($"MaxConnections:  
13.            {_options.MaxConnections}");  
14.    }  
15. }
```

This way, you can access configuration options wherever you need them within your application. Additionally, the ASP.NET Core dependency injection system takes care of resolving dependencies and providing properly instantiated configuration options to the class that needs them.

The Option Pattern in ASP.NET Core provides a clean and robust way to manage application configuration, allowing for greater flexibility and maintainability in the code.

Conclusion

This chapter provided a comprehensive overview of key aspects of web development, particularly within the context of ASP.NET Core projects and in particular focused on Blazor. We have seen the intricate structure of such projects, explored the vital role of middleware and the standard ones, and examined the practical application of basic design patterns applied on a Blazor project.

Armed with insights into project structure, middleware handling, and design patterns, you are well-prepared to delve deeper into the architecture of an ASP.NET Core application, ready to tackle new challenges and explore innovative solutions.

In the next chapter, we will thoroughly explore some of the most widely acclaimed architectural models rooted in fundamental design principles. These include Clean Architecture, which emphasizes the separation of concerns and maintainability; Domain-Driven Design, which centers on aligning software systems with domain complexities; and Model-View-Controller, a pattern facilitating the separation of presentation, business logic, and data handling layers.

Points to remember

- **Structure of the project:** We have three main sections:
 - o Server project
 - o Client project
 - o Location of Blazor JavaScript
- **Middlewares and request pipeline:** There are a lot of built-in middleware ready to include in our projects and specifics for a job.
- **Design patterns:** Design patterns are essential tools in software development that offer proven solutions to common problems.

Multiple choice questions

1. Regarding the Structure of an ASP.NET Core Application

(Blazor WebApp) project, which folder typically contains the client-side code for a Blazor WebAssembly application?

- a. Controllers
 - b. Models
 - c. Pages
 - d. Services
- 2. In the context of Middleware and the request pipeline in ASP.NET Core, which of the following best describes the purpose of middleware?**
- a. Handling database queries
 - b. Authenticating users
 - c. Managing HTTP requests and responses
 - d. Generating HTML templates
- 3. Which design pattern is commonly used in ASP.NET Core applications for managing dependencies and promoting loose coupling between components?**
- a. Singleton Pattern
 - b. Observer Pattern
 - c. Builder Pattern
 - d. Dependency Injection Pattern
- 4. In a ASP.NET Core applications, where is the main entry point typically defined?**
- a. Startup.cs
 - b. Program.cs
 - c. Index.razor
 - d. App.razor
- 5. What role does the UseRouting() method play in the ASP.NET Core request pipeline?**
- a. It handles HTTP requests and responses.
 - b. It configures routing for incoming requests.

- c. It manages database connections.
 - d. It authenticates users.
6. Which folder in an ASP.NET Core project is commonly used to store reusable components and services that can be injected into other parts of the application?
- a. Controllers
 - b. Models
 - c. Services
 - d. Middleware

Answer key

Key terms

- **Middleware:** This software handles requests and responses in the application pipeline.
- **Component-based architecture:** This design approach involves structuring an application as a collection of loosely coupled, reusable components, each responsible for a distinct and modular functionality.
- **Event-driven architecture:** This is an architectural pattern that enables components to react to and handle events, improving responsiveness and scalability of applications.
- **Option Pattern:** It is a way to manage the application starting from the configuration settings, allowing the application to be dynamic and resilient.
- **Command Pattern:** Decouple the requester of an operation from the object that performs the action.

CHAPTER 3

Architectures and Core Components

Introduction

In this chapter, our primary focus will be to acquaint you with some of the most prevalent architectural paradigms rooted in fundamental design principles. Specifically, we will embark on an exploration of Clean Architecture, **Domain-Driven Design (DDD)**, **Hexagonal Architecture (HA)**, and the widely adopted **Model-View-Controller (MVC)** architecture, each representing distinct software design components. These architectural approaches have garnered significant attention and adoption in the software development community due to their emphasis on promoting robust software design practices.

Structure

In this chapter, we will cover the following topics:

- Overview of different architectures
- Understanding Model-View-Controller

- Understanding Clean Architecture
- Understanding Domain-Driven Design
- Understanding Hexagonal Architecture

Objectives

This chapter aims to provide the basic tools needed to build our software as effectively as possible. By examining MVC, HA, CA, and DDD, we will understand how these design principle-based architecture models prioritize crucial aspects such as separation of concerns, modularity, and maintainability. By exploring these architectures, you can equip yourself with essential Core Components and insights necessary for crafting scalable, maintainable, and resilient software solutions that meet the evolving demands of modern applications.

Overview of different architectures

Choosing the right architectural pattern is challenging. The software architect must understand the application's domain, use cases, and relevant technologies to make an appropriate choice. They should also be familiar with key design patterns to optimize and standardize development rules. Software architecture involves the high-level structure of a system, its components, their relationships, and design principles. Different architectures offer various benefits and trade-offs for organizing and designing software systems.

Monolithic architecture

In a monolithic architecture, the entire application is developed and deployed as a single unit. At the beginning of the project, this approach may seem simpler, but as the project evolves and grows, it can become complex and difficult to maintain. All components, that is, business logic, user interface, and data persistence are integrated and run on a single process.

As shown in [Figure 3.1](#), all parts of a monolithic application operate together with some overlapping between layers:

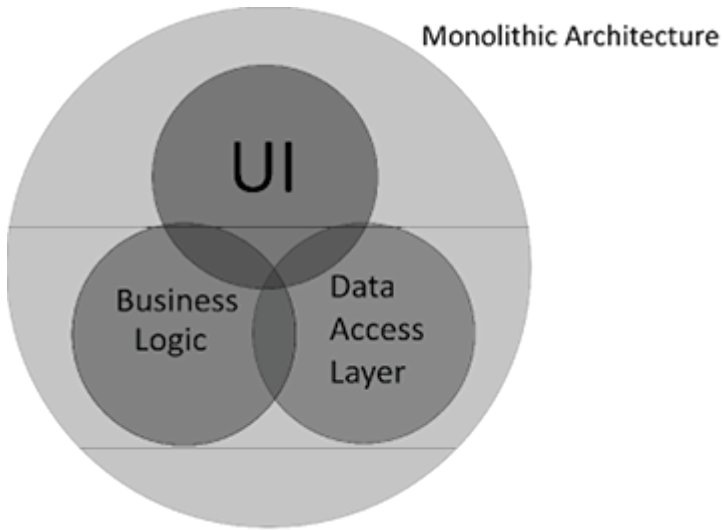


Figure 3.1: Monolithic architecture

There are some advantages and disadvantages to using this approach. The main points of value in a monolithic architecture are as follows:

- **Simplicity:** Since all application components are implemented within a single process, managing and deploying the application can be relatively straightforward.
- **Ease of development:** Monolithic architecture simplifies development as it does not require managing communication between different services or components, does not need synchronization, and, most of all, it does not have problems with out-of-process interactions.
- **Ease of debugging:** Being a single unit, debugging and troubleshooting issues in the application can be simpler compared to distributed architectures.
- **Performance:** In some situations, a monolithic architecture can be more performant than a distributed architecture as there is no overhead due to communication between services. We can think about real-time applications or software that run on our smartwatches.

Monolithic architecture, while offering advantages in certain contexts, is accompanied by numerous drawbacks that deserve to be

taken into consideration before adopting this type of solution. Following are some well-known drawbacks:

- **Limited scalability:** Due to the entire application running as a single unit, scaling specific sections of the application independently can be challenging.
- **Complex maintenance:** Large monolithic applications can become complex to maintain and update over time, especially if the codebase becomes extremely large and intricate.
- **Limited or null resilience:** A failure in one part of the application can impact the entire application, making it difficult to isolate and recover from issues.
- **Technological limitations:** Since the application is developed as a single entity, integrating new technologies or replacing parts of the application with newer technologies can be challenging without significant rewriting.

In a monolithic application, all components, including the user interface, business logic, and data access layers, are tightly coupled and deployed as a single unit. For instance, a monolithic app might use a single C# codebase with ASP.NET Core. The frontend, backend, and database are integrated within the same application. While monolithic architectures were common, there is a trend toward modular, distributed architectures like microservices as applications grow. Monolithic architectures are still suitable for smaller projects where managing separate services is not justified. Ensure your solution is modular to allow scaling with changing business needs and requirements.

Microservices architecture

Microservices architecture decomposes an application into a set of small, independent services, each responsible for a specific function. This promotes scalability and flexibility but requires careful management of inter-service communication.

Refer to the following figure for a better understanding:

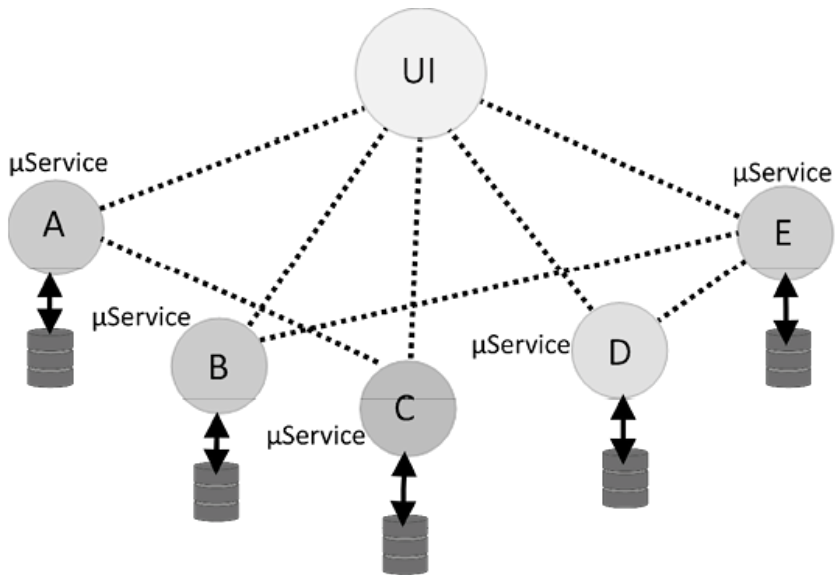


Figure 3.2: Microservice architecture

Microservices architecture decomposes an application into discrete services, each dedicated to a specific function. These services interact via a service broker or sockets and may have their own databases. The key features are as follows:

- **System decomposition:** Divides a monolithic application into smaller, well-defined components.
- **Autonomy and scalability:** The services operate independently, allowing separate development, testing, and deployment, and enabling scalability based on workload.
- **Heterogeneous technologies:** Its services can use different technologies as long as they are interoperable via APIs or service brokers.
- **Fault isolation:** The failures in one service do not affect others.
- **Organizational scalability:** This supports distributed, autonomous teams for greater agility.
- **Evolved development:** Its modular design enables rapid development and updates without impacting other services, as long as interfaces remain consistent.

On the other hand, there are also drawbacks to using this type of

architecture, some of which are as follows:

- **Operational complexity:** Managing a multitude of microservices can increase operational complexity, requiring additional tools and skills.
- **Infrastructure overhead:** Implementing microservices can incur additional costs and require resources for supporting infrastructure.
- **Transition complexity:** Migrating from a monolithic architecture to one based on microservices can be complex and require significant time and resources.

In summary, microservices applications offer a range of advantages, including greater flexibility, scalability, and resilience, making them particularly suitable for complex and distributed systems. However, it is important to carefully consider the associated trade-offs, such as management complexity and the need to handle communication between services, before adopting this architecture.

Serverless architecture

Serverless architecture, also known as **Function as a Service (FaaS)**, is a cloud computing model where cloud providers manage the infrastructure required to run and scale applications. In a serverless architecture, developers can focus solely on writing code for their application's functionality without worrying about provisioning or managing servers.

Following is the base model of a serverless architecture:

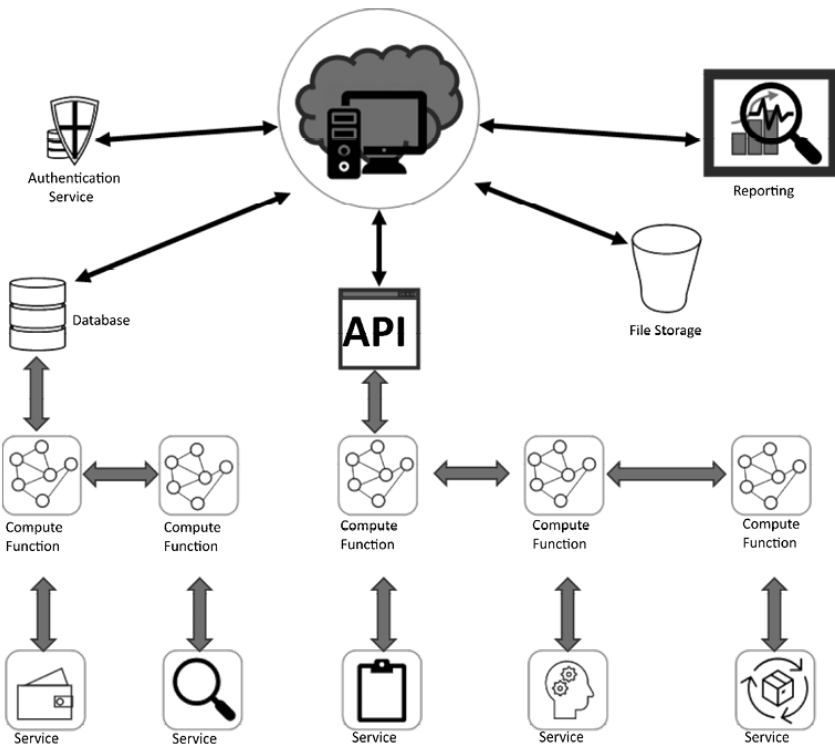


Figure 3.3: *Serverless architecture*

In a serverless architecture, functions are triggered by events like HTTP requests, database changes, or file uploads. These functions execute tasks only when needed, with cloud providers dynamically allocating resources, leading to cost savings as you only pay for actual usage. Serverless is particularly cost-effective for infrequent operations and scales automatically based on incoming events. Functions are stateless, meaning they do not maintain persistent state between invocations, requiring any necessary state to be stored externally. Serverless applications often integrate with third-party services, such as databases and authentication services, to perform tasks beyond individual functions' scope. The huge success of the serverless architecture is also due to the current cloud platforms that host our applications in the form of functions with a series of significant advantages.

Some of the benefits are as follows:

- **Scalability:** Applications scale automatically to handle

varying workloads.

- **Cost-effectiveness:** Pay-per-use billing reduces costs for sporadic applications.
- **Simplified operations:** Developers focus on application logic, reducing infrastructure management.
- **Faster time-to-market:** Developers quickly deploy and iterate applications without infrastructure concerns.

However, serverless architecture also comes with some challenges, such as cold start latency, vendor lock-in, limited runtime environments, resource constraints, debugging and monitoring complexity, security concerns, cost management, limited control and customization, and operational complexity.

Overall, serverless architecture provides a flexible and efficient way to build and deploy applications, particularly for use cases with unpredictable workloads or short-lived tasks.

Event-driven architecture

Event-driven architecture, also known as **Publisher/Subscriber**, is a design pattern for building applications that respond to and process events. Widely used in microservices, it also facilitates communication among multiple monoliths. In this architecture, data flow and control are determined by asynchronous events, which can include user interactions, system notifications, or messages from other systems. Events are core entities, representing significant occurrences within the system or external environment.

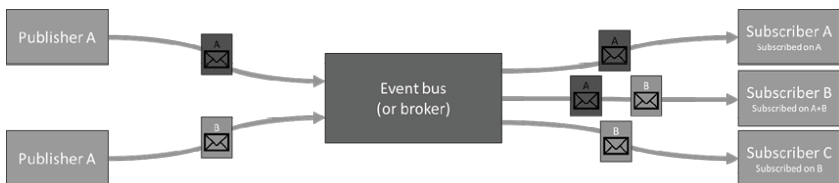


Figure 3.4: Event-driven architecture

There are three key components of event-driven architecture, as follows:

- **Event sources:** These are the components or systems that generate events (publishers). These could be user interfaces,

sensors, databases, or external systems. Event sources publish events to the event-driven system.

- **Event bus or broker:** This is the backbone of the event-driven architecture. It is responsible for routing events from event sources to event consumers. The event bus typically decouples event sources from event consumers, allowing for scalability, flexibility, and loose coupling between components.
- **Event consumers:** They are the components or systems that react to events (subscribers). They subscribe to specific types of events and perform actions in response to those events. Event consumers can be other services, modules, or processes within the same system or external systems.

Handling events within an event bus involves filtering, transforming, and delivering them to consumers using rules engines, stream processing, or complex event processing. A key advantage is decoupling system components, which enhances scalability, flexibility, and independent evolution. Components communicate asynchronously, improving responsiveness and performance. This architecture enhances fault tolerance, as failures of any subscriber or publisher do not impact the entire system. Event-driven systems are ideal for reactive, scalable, and resilient applications, efficiently processing large volumes of real-time events. They are widely used in IoT, financial services, e-commerce, and telecommunications.

Design principle-based architectures

Design principle-based architectures group architectural approaches or frameworks that are built around a set of guiding design principles. These principles provide a foundation for making architectural decisions and designing systems that are aligned with desired characteristics such as scalability, maintainability, reliability, and performance.

Architectures like Clean Architecture, DDD, and HA emphasize principles such as separation of concerns, modularity, and maintainability. They provide guidelines for structuring software systems to enhance comprehensibility and ease of modification.

Design principles are high-level guidelines or concepts that capture

best practices and established patterns for designing software systems. Design principles are as follows:

- **Modularity:** Design systems with independent modules for separate development, testing, and maintenance.
- **Separation of concerns:** Separate system aspects (presentation, business logic, data storage) for clarity and maintainability.
- **Abstraction:** Hide complex implementation details behind simple interfaces for higher-level interaction.
- **Encapsulation:** Bundle data and methods into a unit, controlling access to prevent unintended modification.
- **Decoupling:** Minimize dependencies to reduce impact of changes and enhance flexibility.
- **Scalability:** Design systems to scale by adding resources or distributing load.
- **Resilience:** Build systems to recover gracefully from failures and maintain functionality.
- **Performance:** Optimize system performance with efficient algorithms, data structures, and resource utilization.
- **Simplicity:** Favor simple designs to enhance understandability and maintainability.
- **Flexibility:** Design systems to adapt to changing requirements without major redesigns.

Architectures based on principles like MVC, Clean Architecture, DDD, and HA are robust, flexible, and easier to maintain. They guide design and development decisions, helping teams build systems that meet current and future needs effectively. Each architecture has its advantages and challenges, with the choice depending on project requirements, scalability, and team expertise. Understanding these architectures allows software engineers to design systems tailored to their specific goals and constraints. The choice of architecture should consider project needs, functional and non-functional requirements, business goals, and technological constraints.

Understanding Model-View-Controller

MVC architecture in ASP.NET Core divides an application into three interconnected components: the Model, the View, and the Controller. The Model represents the application's data and business logic, the View displays the data to the user, and the Controller handles user input and updates the Model accordingly. This architectural pattern promotes separation of concerns, making it easier to maintain and scale ASP.NET Core applications.

The most used architectural pattern in ASP.NET Core is essentially MVC because it is natively supported. It comes from a long time ago when ASP.NET (in the .NET framework version 3.5 in 2009) introduced the MVC pattern natively.

Following figure shows the main diagram of MVC pattern:

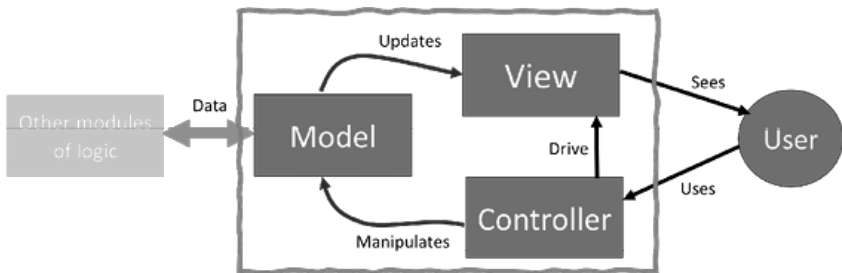


Figure 3.5: MVC architecture

MVC pattern divides an application into the following three main components:

- **Model:** Represents data structure, contains business logic, ensures consistency.
- **View:** Manages **user interface** (UI), presents data from the Model comprehensibly and interactively to users.
- **Controller:** Receives input, manages control flow, translates actions, and coordinates Model and View interactions.

The goal of the MVC pattern is to separate responsibilities within the application, making the code more modular, maintainable, and scalable. This separation also enables greater code reusability and helps manage the complexities of the application, improving code clarity and structure.

MVC pattern

MVC is often considered both a design pattern and an architectural pattern, depending on the context in which it is used. Refer to the following points to understand the pattern:

- As a design pattern, MVC describes the separation of concerns within a single module or component. It divides an application into three interconnected components: Model (data and business logic), View (user interface), and Controller (handles user input and updates the Model and View accordingly). This separation helps in achieving modularity and maintainability within a single application module.
- As an architectural pattern, MVC is also applied at a higher level, defining the overall structure and organization of an entire application. In this context, MVC helps in organizing the codebase of the entire application by dividing it into distinct components with specific responsibilities. This promotes clean and maintainable architecture.

For this book, we consider MVC as an architectural pattern.

Implementation

ASP.NET Core implements a template that corresponds to MVC, the template (as we have seen in [Chapter 1, Introduction to ASP.NET Core](#), named *ASP.NET Core Web App (Model-View-Controller)*). The structure of this template is relatively easy to understand.

Refer to the following figure to see what it looks like:

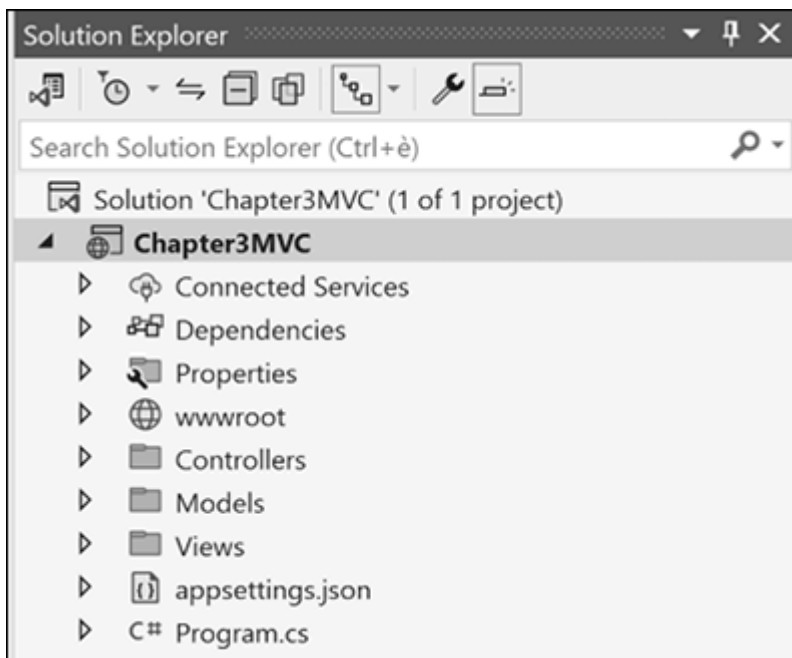


Figure 3.6: VS Project on ASP.NET Core Web App (Model-View-Controller) template

In this case, it is easy to understand which kind of separation was made. Each folder in the project tree represents one of the components involved in the architecture pattern. The running application is quite simple.

Following is the main screen of our sample application:

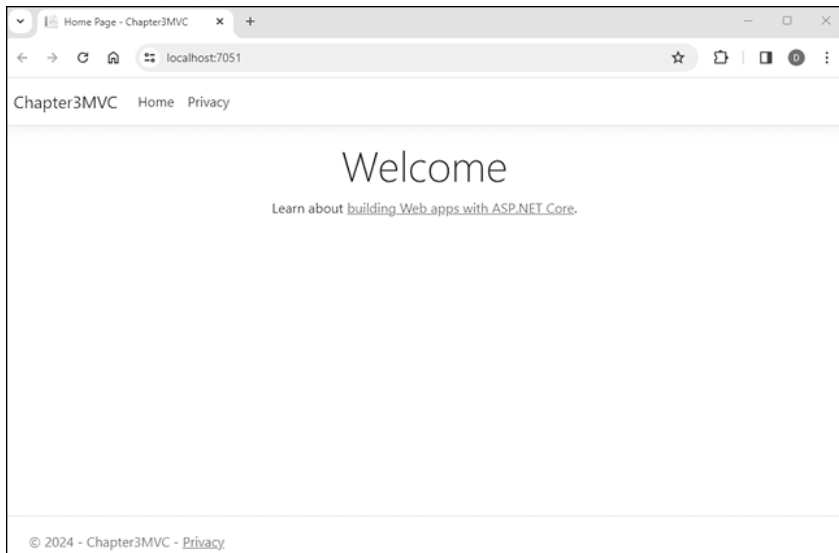


Figure 3.7: Running application

Now, we can imagine modifying this basic template according to our needs. For instance, we can create a simple application that allows the user to see his task list and manage it. For this example, we will not use any database, but it would be possible.

Let us see the main steps to achieving the goal as follows:

1. For the first step, we have to add the model class we will use for adding new tasks. In the folder named *Models*, add a class C# with the following content:
 1. `public class TaskModel`
 2. `{`
 3. `public int Id { get; set; }`
 4. `public string Description { get; set; } = string.Empty;`
 5. `public bool IsCompleted { get; set; }`
 6. `}`
 7.
2. Now, let us modify the **Index.cshtml** page. We have to add a table that will show the list of our tasks. The code can be accessed from the github link [The code can be accessed from the github link.](#)
3. The next step is a bit more complicated. We are going to

build a **Create, Read, Update, Delete (CRUD)** application. We need two more pages: **Create.cshtml** and **Edit.cshtml**, but no delete page as we will delete each task directly.

We need a controller to manage API calls. Understanding REST APIs in depth will be covered in [Chapter 4, Designing RESTful APIs](#).

Our controller, named **HomeController**, will manage a task list with CRUD operations. It handles task creation, editing, and deletion. The Index action displays tasks, while Create and Edit handle task addition and modification. The Delete function removes tasks directly.

In our controller, we will craft two Create methods and two Edit methods. The first Create method, which returns **View()**, drives the UI via the **Create.cshtml** page. The second method, requiring a **TaskModel** as an argument, adds the task to the task list and navigates back to the home page. For Edit, a method accepting a numeric ID creates a view from **Edit.cshtml**, and another method with a **TaskModel** parameter saves changes and returns to the home page.

The result will be as follows:

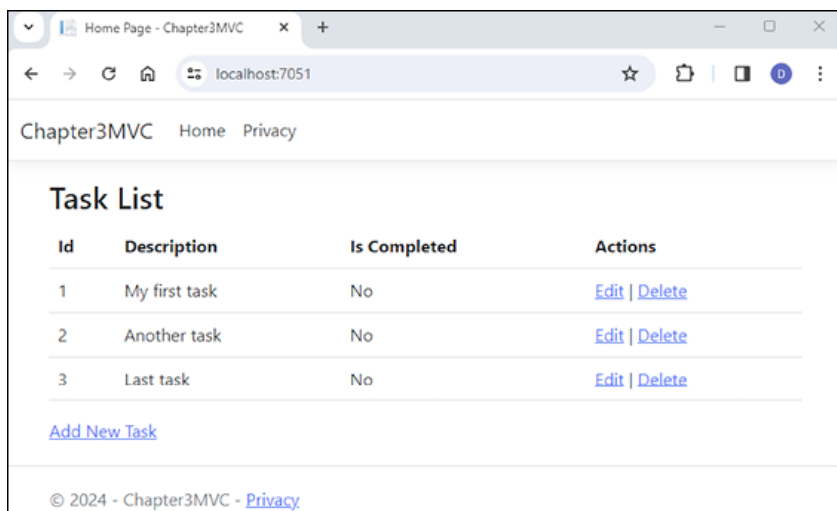


Figure 3.8: Project Task List running

MVC stands as a robust architecture and ASP.NET Core offers the ideal framework for harnessing this pattern efficiently.

Understanding Clean Architecture

Another architectural pattern used in ASP.NET Core application is Clean Architecture. It is a software development approach proposed by *Robert C. Martin*, aiming to create modular, framework-independent, and easily testable systems. The fundamental concept of Clean Architecture is to separate software concerns into concentric circles, with each circle representing a level of abstraction and responsibility.

Refer to the following figure for a better understanding:

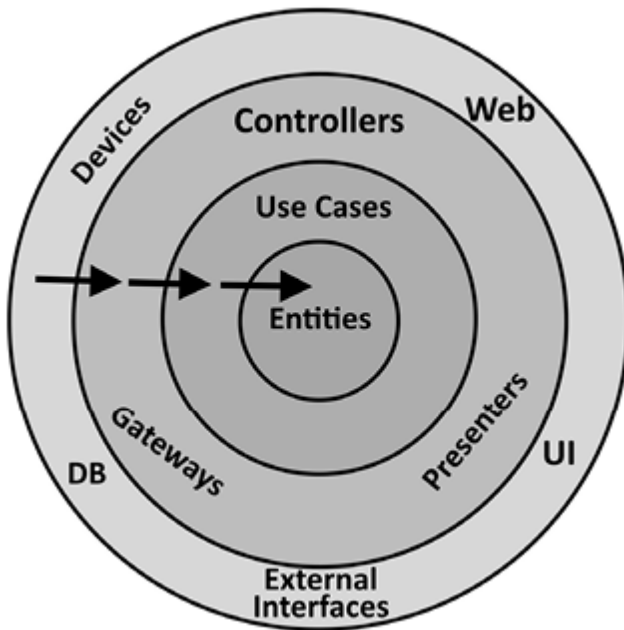


Figure 3.9: Clean Architecture¹

The concentric circles represent different areas of software. In general, the further you go, the higher level the software becomes. The outer circles are mechanisms and the inner circles are policies.

We can translate the words *Uncle Bob*, also known as *Robert C. Martin*, into the following key concepts of Clean Architecture:

- **Entities:** These are basic entities, primary objects, often corresponding to domain concepts.

- **Use cases:** Encapsulates business logic, defines behaviors, is framework-independent, and contains business rules.
- **Interface adapters:** These bridge internal and external circles, converting data for use cases and external infrastructure (for example, databases, frameworks, and user interfaces).
- **Frameworks and drivers:** These outer layers contain specific implementation details like databases or presentation frameworks.

Clean Architecture promotes dependency inversion, ensuring inner layers are unaware of outer details. This separation of concerns enhances maintenance, testability, and scalability. By dividing software into layers and adhering to the Dependency Rule, we can create an easily testable system. In Clean Architecture, the Dependency Rule states that source code dependencies must always point inward (respect to the circular diagram shown in [Figure 3.9](#)), toward higher-level policies, meaning that inner layers (entities, use cases) are independent of outer layers (UI, database, frameworks). Outdated components like databases or frameworks can be swapped out without significant hassle, adapting to future changes smoothly.

When Clean Architecture fits and fails

Clean Architecture is ideal for complex projects needing maintainability, scalability, and testability. It structures codebases for large-scale applications, promoting long-term viability and ease of collaboration. With a clear separation of concerns and minimized dependencies, it reduces technical debt, making future changes or feature additions simpler.

However, Clean Architecture is not always the best choice. For simple applications, it can add unnecessary complexity. Strict deadlines and lack of team expertise can make its implementation impractical. Retrofitting onto legacy systems is challenging, and rapid prototyping often benefits from simpler architectures. Smaller teams might find the overhead excessive compared to its benefits.

Clean Architecture in practice with ASP.NET Core project

We will explore how to implement the principles of Clean

Architecture in a Blazor web application. We will use a practical example to structure a Blazor project following these principles. By developing a task list application, we will compare it to MVC, highlighting the main differences and benefits of Clean Architecture.

Without further ado, let us proceed with the implementation of our example, exploring the details of how Clean Architecture can be successfully applied in a Blazor context.

As we have seen before, the main parts of a Clean Architecture are *Entities*, *Use Cases*, *Interfaces*, *Adapters*, and *Frameworks and Drivers*.

According to this framework, we can hypothesize a project structure, based on the Blazor WebApp template. The template provides two projects as we have seen in [Chapter 1, Introduction to ASP.NET Core](#), the base project *server side*, and the project *client side* names **.Client**.

The two projects, within the framework of Clean Architecture, should be the **Presentation** (or simply the UI) part. It fits the outer ring in [Figure 3.9](#) containing *devices*, *web*, *DB*, *external devices*, and *UI*.

In our implementation, we can figure out the following main parts:

- **Infrastructure:** It represents interfaces through the external parts as adapters. This part fits the *Controllers/Gateways/Presenters* in the [Figure 3.9](#).
- **Domain:** It contains use cases, events, repositories (see section *Repositories and repository pattern*). This part fits the *Use cases* in [Figure 3.9](#).
- **Core:** It will contain business rules, enums, exceptions, interfaces, and entities. This part fits the *Entity* in [Figure 3.9](#).

In our project, we will add new projects as *Razor Class library*, and our new structure of the project will be as follows:

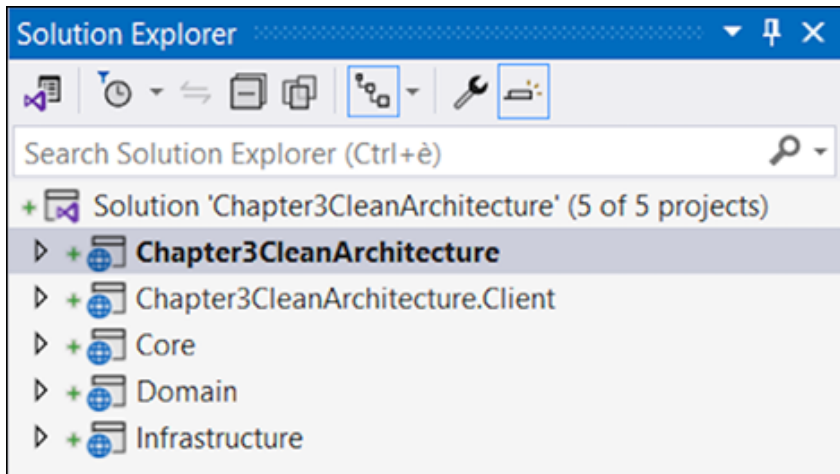


Figure 3.10: Clean Architecture project structure

Now, we can start to structure our Task List example. The goal of this example will be the same as the previous chapter but with a different architecture.

In order to start with our Task List, we can reuse the same model **TaskModel**, and put it in the project named **Core**. For the following example, we will copy and paste the same code, but we will rename it (only to be coherent) as **TaskEntity**:

```
1. public class TaskEntity
2. {
3.     public int Id { get; set; }
4.     public string Description { get; set; } = string.Empty;
5.     public bool IsCompleted { get; set; }
6. }
7.
```

Now that we have to structure a place to store all the Tasks of the application, the right place, according to our project organization, is the **Domain** project, in particular, we made a new folder called **Repository** where we put a new class as follows:

```
1. namespace Domain.Repositories;
2. public class TaskEntities : ITaskEntities
3. {
4.     List<TaskEntity> entities = [];
5.     public void AddEntity(TaskEntity entity)
```

```

6.     => entities.Add(entity);
7.     public IEnumerable<TaskEntity> GetEntities() => entities;
8.     public TaskEntity? GetEntityFromId(int id)
9.     => entities.FirstOrDefault(x => x.Id == id);
10.    public void RemoveEntity(TaskEntity entity)
11.    {
12.        if (entity is not null)
13.            entities.Remove(entity);
14.    }
15. }

```

We also need to define the interface **ITaskEntities** in the **Core** project into an **Interface** folder by using the following code:

```

1. using Core.Entities;
2. namespace Core.Interfaces;
3. public interface ITaskEntities
4. {
5.     List<TaskEntity> Entities { get; set; }
6. }

```

The place where our tasks will be stored is ready and we can use it in the whole application. To make it real, we have to declare it in our Dependency Injection. So, in the main project, we add the following line in the **Program.cs**:

```
1. builder.Services.AddSingleton<ITaskEntities, TaskEntities>();
```

After this, we can use **TaskEntities** in the presentation layer. At this point, the application needs two other pages to properly manage our tasks (as a CRUD application) **Create.razor** and **Edit.razor**, and a Home page to present data. The code can be accessed from the github link The code can be accessed from the github link.

The last function we need for this application is the delete. For this application, we can delete the entity directly from the **Home.razor** by adding the method delete as shown in the following code:

```
1. <a href="#" @onclick="(() => Delete(task.Id))">Delete</a>
```

The C# method will be as follows:

```

1.     private void Delete(int id)
2.     {
3.         var taskElement = Repository.Entities.FirstOrDefault(e =>

```

```
e.Id == id);  
4.     if(taskElement is not null)  
5.         Repository.Entities.Remove(taskElement);  
6.     }  
7. }
```

As we can see, the functions will be simple, but the underlying code is quite complex.

Summarizing up Clean Architecture

It is evident from the code in this example that adopting Clean Architecture demands considerable effort. This architectural approach prioritizes meticulous planning and structured organization, making it particularly well-suited for substantial projects seeking a coherent division of components. Its implementation requires a comprehensive understanding of system design principles, as well as a commitment to maintaining code cleanliness and scalability. Therefore, Clean Architecture emerges as a strategic choice for enterprises navigating complex development landscapes, where clarity, maintainability, and scalability are important.

Understanding Domain-Driven Design

DDD is not directly linked to the code but is a software development approach that focuses on understanding and modeling the domain of an application. Created by *Eric Evans*, DDD emphasizes collaboration between domain experts (such as business stakeholders) and developers to create a shared model of the application's domain. From a developer's perspective, the code structure resembles Clean Architecture. However, for software architects in DDD, it differs. Architects define the system architecture to reflect business domain concepts, working closely with domain experts. They identify and define bounded contexts to manage complexity and ensure clear responsibilities. Architecture is driven by domain characteristics, with design decisions based on domain and business needs rather than technological considerations.

Ubiquitous Language

The system architects find themselves facing the **Ubiquitous Language**. This concept promotes the use of a common language between developers and domain experts. The language used in the code should reflect the language used by business stakeholders, creating a shared understanding. Ubiquitous Language bridges technical and non-technical stakeholders by establishing a shared vocabulary and fostering effective communication and collaboration. This common understanding minimizes misunderstandings, ensuring the code aligns with professional language, ultimately reducing misinterpretations during development. When everyone uses the same terms, misunderstandings are minimized.

Another key concept of DDD is the **Bounded Context**. It defines clear boundaries within which a particular model is valid. Within a bounded context, a term or concept may have a specific meaning, but that meaning might be different in another bounded context. In other words, split the ubiquitous language into several smaller languages, and then allocate each one to the specific context where it is applicable.

In essence, a bounded context defines the boundaries within which a certain model is valid and meaningful. *What changes together is within the same boundary.* This clarity prevents ambiguity and misunderstandings, ensuring team members share a common understanding. Assigning specific teams to each bounded context fosters specialization and better communication.

Well-defined bounded contexts enhance understanding of the problem domain, facilitate modular software development, and improve team and change management over time.

Entities and value objects

In DDD, entities are distinct objects within the domain model, encapsulating data and behavior, with a unique identity for consistent reference and tracking. They maintain a coherent state throughout their lifecycle and enforce business rules, promoting encapsulation. Entities define relationships, participating in aggregates or hierarchies reflecting the business domain.

Value objects in DDD are immutable and represent significant domain aspects without unique identities. They describe the characteristics or properties of entities. Two value objects with the same state are equal and interchangeable. Their identity derives solely from their attribute values, unlike entities which have a unique identity, value objects derive their identity solely from their attribute values.

Take a look at the following example for a better understanding:

```
1. using System;
2. // Value Object: EmailAddress
3.
4.     EmailAddress     email1         =     new
EmailAddress("example@email.com");
5.     EmailAddress     email2         =     new
EmailAddress("example@email.com");
6.
7. Console.WriteLine(email1.Equals(email2)); // Output: True
8.
9. // Entity: Person
10. Person person1 = new Person("Alice", 30);
11. Person person2 = new Person("Alice", 30);
12.
13. Console.WriteLine(person1.Equals(person2)); // Output: False
14.
15.
16. // Value Object: EmailAddress
17. public class EmailAddress
18. {
19.     public string Address { get; }
20.
21.     public EmailAddress(string address)
22.     {
23.         Address = address;
24.     }
25.
26.     public override bool Equals(object obj)
27.     {
28.         return obj is EmailAddress other && Address ==
```

```

other.Address;
29.  }
30.
31.  // Override GetHashCode if overriding Equals
32.  public override int GetHashCode()
33.  {
34.      return Address.GetHashCode();
35.  }
36. }
37.
38. // Entity: Person
39. public class Person
40. {
41.     public int Id { get; set; }
42.     public string Name { get; }
43.     public int Age { get; }
44.
45.     public Person(string name, int age)
46.     {
47.         Name = name;
48.         Age = age;
49.     }
50.
51.     public override bool Equals(object obj)
52.     {
53.         return obj is Person other && Id == other.Id;
54.     }
55.
56.     // Override GetHashCode if overriding Equals
57.     public override int GetHashCode()
58.     {
59.         return Id.GetHashCode();
60.     }
61. }
62.

```

In this simple example, the distinction between a *value object* and an *entity* becomes evident.

The **EmailAddress** class represents a *value object*. *Value objects* are characterized by their immutable nature and equality based on their attribute values. In this case, an email address encapsulated by the **EmailAddress** class is considered a *value object* because it is defined by its value and does not have its own identity. Two **EmailAddress** instances with the same email address value are considered equal, demonstrating the value-based semantics of the *value object*.

On the other hand, the **Person** class represents an *entity*. *Entities* have their own distinct identity, typically represented by a unique identifier. In this example, the **Person** class has an **Id** property that uniquely identifies each instance. *Entities* are compared based on their identity rather than their attribute values. Even if two **Person** instances have the same name and age, they are not considered equal unless they share the same identity, highlighting the identity-aware nature of *entities*.

This example showcases the fundamental difference between *value objects* and *entities* in domain modeling: *value object* emphasizes value-based equality and immutability, while *entities* focus on identity and mutability. Understanding this difference is crucial for designing effective domain models in software systems.

Aggregates

Aggregates are groups of related *entities* and *value objects* treated as a cohesive unit. An *entity* within an aggregate is the aggregate root, responsible for ensuring consistency of operations on *entities* within the aggregate. Let us see a simple example, as follows:

```
1. public class Order
2. {
3.     public int Id { get; private set; }
4.     private readonly List< OrderLine > orderLines = new();
5.     public IEnumerable< OrderLine > OrderLines =>
        orderLines.AsEnumerable();
6.
7.     // Add an order line to the order
8.     public void AddOrderLine(OrderLine orderLine)
9.     {
10.         orderLines.Add(orderLine);
```



```

11.     }
12. }
13.
14. public class OrderLine
15. {
16.     public int Id { get; private set; }
17.     public string ProductName { get; private set; }
18.     public decimal Price { get; private set; }
19.     public int Quantity { get; private set; }
20.
21.     public OrderLine(string productName, decimal price, int
quantity)
22.     {
23.         ProductName = productName;
24.         Price = price;
25.         Quantity = quantity;
26.     }
27.
28.     // Calculate the total price for this order line
29.     public decimal CalculateTotalPrice() => Price * Quantity;
30. }

```

In this example, the aggregate root is represented by the class **Order**, that ensures consistency and integrity within the *aggregate*. All operations on **OrderLines** should go through the order aggregate root (**Order**) to maintain data integrity and encapsulation.

Repositories and repository pattern

A *repository*, in DDD, provides an interface to access and persist domain objects. *Repositories* help isolate domain code from persistence logic. The optimal approach is to illustrate the repository with an example. According to the example seen in the Clean Architecture paragraph, we can figure out how to replace the existing *domain* class able to store the list of tasks.

Before we see how to improve the implementation of the location to store our tasks, we must see how this design pattern works.

The **repository pattern** is a common design pattern used to separate data access logic from business logic within an application. This pattern is widely used in software systems to promote

modularity, maintainability, and testability of the code.
Refer to the following figure:

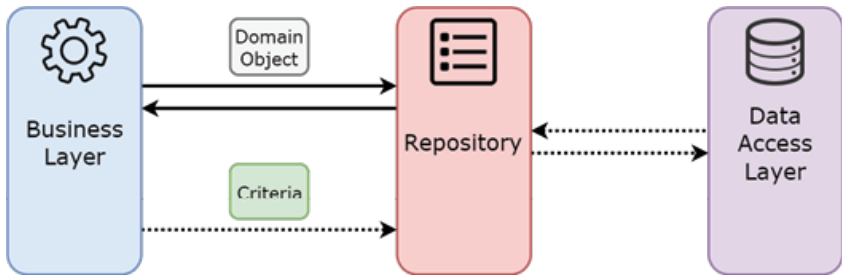


Figure 3.11: Repository pattern

The repository pattern provides an interface for CRUD operations, separating business from data access logic, and enhancing code modularity and testability. It allows swapping data layers (for example, from relational to NoSQL) without altering business logic. Concrete implementations interact with various persistence systems, enabling focused and less infrastructure-dependent testing.

So, for our purpose in the **TaskList** example, at the beginning, we have to define an interface in the Core project, refer to the following code for the same:

```
1. namespace Core.Interfaces;
2. public interface IRepository
3. {
4.     TaskEntity? GetById(int id);
5.     IEnumerable<TaskEntity> GetAll();
6.     void Add(TaskEntity entity);
7.     void Update(TaskEntity entity);
8.     void Delete(TaskEntity entity);
9. }
```

Then, we provide a concrete implementation of it in the Domain project as shown in the following code:

```
1. using Core.Entities;
2. using Core.Interfaces;
3. namespace Domain.Repositories;
4. public class TaskEntityRepository : ITaskEntityRepository
5. {
```

```

6.  private readonly List<TaskEntity> entities =
7.      new List<TaskEntity>();
8.  public TaskEntity? GetById(int id)
9.      => return entities.FirstOrDefault(e => e.Id == id);
10.
11. public IEnumerable<TaskEntity> GetAll() => entities;
12. public void Add(TaskEntity entity)
13.     => entities.Add(entity);
14. public void Update(TaskEntity entity)
15. {
16.     var existingEntity = entities.FirstOrDefault
17.         (e => e.Id == entity.Id);
18.     if (existingEntity is not null)
19.     {
20.         var index = entities.IndexOf(existingEntity);
21.         entities[index] = entity;
22.     }
23. }
24. public void Delete(TaskEntity entity) =>
25.     entities.Remove(entity);
26. }

```

With some little modification on our presentation layer, we can have a fully working repository pattern. In the future when we use a database to persist data or (for instance) use the local storage of the browser, we have only to implement and use another instance of the interface **IRepository**.

Services

In DDD, services are key components that encapsulate domain operations or logic not specific to any entity or value object. They coordinate actions across multiple entities, acting as orchestrators for complex workflows. Services promote cleaner separation of concerns, enhance clarity and maintainability, and improve reusability and composability of domain behavior by centralizing logic in focused components.

Domain events

Domain events represent significant occurrences within a system's

domain, acting as key milestones with substantial relevance to its operation. They encapsulate important changes or transitions and trigger responsive actions. Beyond documentation, domain events preserve coherence and synchrony across disparate components. By broadcasting these events, the system ensures effective communication of updates, fostering unified understanding and seamless interactions. *Domain events* maintain systemic integrity and coherence, enabling the system to adapt and respond appropriately to evolving circumstances within its operational domain.

Summing up DDD

The main goal of DDD is to create a domain model reflecting the shared understanding between developers and domain experts, focusing on business needs. It complements Clean Architecture for robust, domain-centered software design.

Understanding Hexagonal Architecture

Hexagonal Architecture, also known as Ports and Adapters Architecture, is a software design pattern that emphasizes the separation of concerns and isolation of components within a system. It was introduced by *Alistair Cockburn* in 2005.

The key idea behind the HA is to decouple the core business logic (the application's domain model) from the external dependencies such as databases, user interfaces, external services, etc. This decoupling is achieved by organizing the components of the system into concentric layers or hexagons, with the core business logic residing at the center.

In the following schematic, we can identify some key components:

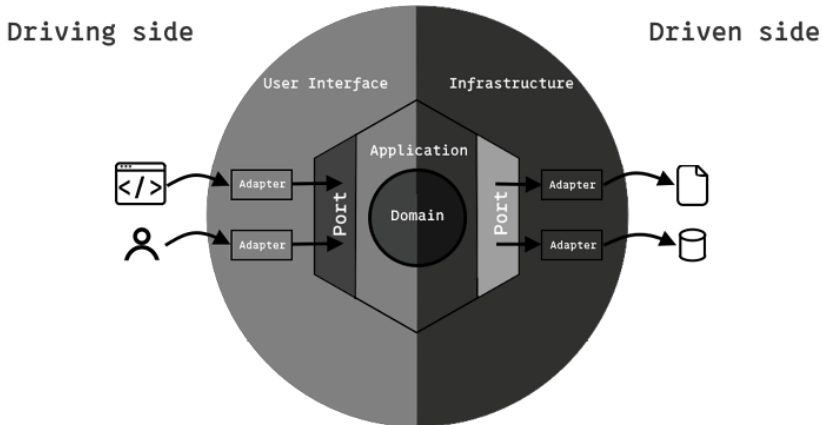


Figure 3.12: Hexagonal Architecture

Let us go through the following key points:

- **Domain:** This is the heart of the application, containing the domain model, business rules, and use cases. It represents the application's unique behavior and is independent of any external concerns.
- **Ports:** They define interfaces through which the core interacts with the external world. They also represent the application's capabilities or services. Ports are typically defined as interfaces or abstract classes without any implementation details. There are usually two types of ports, as follows:
 - **Inbound ports (primary ports):** These are the input mechanisms for external actors (users, systems, devices) to interact with the application, such as REST APIs, message queues, and **graphic user interface (GUI)** interfaces.
 - **Outbound ports (secondary ports):** These ports enable the application to interact with external resources like databases, APIs, and file systems.
- **Adapters:** They are the implementations of the ports. They bridge the gap between external business logic and external dependencies. Adapters convert the data from the format understood by the external system to the format expected by

the core, and vice versa. There can be multiple adapters for each port, allowing the application to interact with different types of external systems without changing the core logic.

HA offers benefits such as **testability** by decoupling core logic from dependencies and facilitating isolated unit tests. Its **flexibility** allows easy swapping or upgrading of external dependencies. The clean separation of concerns enhances **maintainability**, and decoupling core logic ensures portability across various environments or platforms.

Project structure

The essence of *Hexagonal Architecture* lies in its structure. The core logic of the application resides at the center, surrounded by layers of adapters that communicate with the external world. Let us outline the project structure for our ASP.NET Core application, as follows:

- **Domain layer:** This core layer contains domain logic and business rules, framework-independent, housing models, and use cases for application and infrastructure.
- **Application layer:** This layer serves as the intermediary between the *domain layer* and the *user interfaces*. It contains *adapters* that orchestrate business logic and use cases by utilizing *ports*.
- **Infrastructure layer:** This layer implements *adapters* and *ports* to interact with databases, file systems, or third-party services, used by *application layer*.
- **ASP.NET Core layer:** This layer acts as one of the adapters in HA, handles HTTP requests, serves responses, communicates with the *application layer*, and executes business logic.

Implementing Hexagonal Architecture

With a foundational understanding of hexagonal architecture, we will implement it in an ASP.NET Core application. Using a Blazor WebApp template, we will create two projects: *Chapter3HA* and *Chapter3HA.Client*. Refer to [Chapter 2, Basics of ASP.NET Core](#) for structure. Additional Razor Class libraries will be integrated. This setup ensures clear separation of concerns, enhancing

maintainability and flexibility.

Domain layer

As we described in the previous paragraph, the first layer of the structure should be named **Domain**. The Domain Layer represents the application's business logic.

The model will look similar to the following example:

```
1. namespace Domain.Models;
2. public class TaskModel
3. {
4.     public int Id { get; set; }
5.     public string Name { get; set; }
6.     public bool IsCompleted { get; set; }
7. }
```

Application layer

The application layer contains the implementation of use cases or application services that orchestrate the business logic. In our case, we have to make two following folders in the razor project:

- Interfaces
- Adapters

In the **Interface** folder, we create a new interface named **ITaskManagementPort.cs** which will act as a **Port**. Refer to the following code for a better understanding:

```
1. using Domain.Models;
2. namespace Application.Interfaces;
3. public interface ITaskManagementPort
4. {
5.     IEnumerable<TaskModel> GetAllTasks();
6.     void AddTask(TaskModel task);
7.     void UpdateTask(TaskModel task);
8.     void DeleteTask(int taskId);
9. }
```

In the **Adapters** folder, we will create the implementation of this interface as a new class named **TaskManagementAdapter.cs** which will act as **Adapter**. The class will look similar to the following code:

```

1. namespace Application.Models;
2. using Application.Interfaces;
3. using Core.Models;
4. using Infrastructure.Interfaces;
5. using System.Collections.Generic;
6. public class TaskManagementAdapter : ITaskManagementPort
7. {
8.     private readonly ITaskPort adapter;
9.     public TaskManagementAdapter(ITaskPort adapter)
10.     => this.adapter = adapter;
11.     public IEnumerable<TaskModel> GetAllTasks()
12.     => adapter.GetAllTasks();
13.     public void AddTask(TaskModel task)
14.     => adapter.AddTask(task);
15.     public void UpdateTask(TaskModel task)
16.     => adapter.UpdateTask(task);
17.     public void DeleteTask(int taskId)
18.     => adapter.DeleteTask(taskId);
19. }

```

Infrastructure layer

the infrastructure layer contains implementations of ports as **Interfaces** and **Adapters** as classes interacting with external systems. In our example we do not have external systems as databases or calls to external Web API but only a list of tasks that act as a repository in memory. However, we would like to reiterate that the choice was made only for the brevity of the treatment of the topic. So, in this project too we will have two folders, Interfaces and Adapters.

In the following example, we have an interface named **ITaskPort.cs** and its implementation named **TaskAdapter.cs**:

```

1. using Domain.Models;
2. namespace Infrastructure.Interfaces;
3. public interface ITaskPort
4. {
5.     IEnumerable<TaskModel> GetAllTasks();
6.     void AddTask(TaskModel task);
7.     void UpdateTask(TaskModel task);

```



```

8.     void DeleteTask(int taskId);
9. }
1. public class TaskAdapter : ITaskPort
2. {
3.     private static List<TaskModel> tasks = new
List<TaskModel>();
4.     public IEnumerable<TaskModel> GetAllTasks() => tasks;
5.     public void AddTask(TaskModel task) => tasks.Add(task);
6.     public void UpdateTask(TaskModel task)
7.     {
8.         if (tasks.Any(x => x.Id == task.Id))
9.         {
10.             var updateTask = tasks.Where
11.                 (x => x.Id == task.Id).FirstOrDefault();
12.             if (updateTask is not null)
13.             {
14.                 updateTask.IsCompleted = task.IsCompleted;
15.                 updateTask.Name = task.Name;
16.             }
17.         }
18.     }
19.     public void DeleteTask(int taskId)
20.     {
21.         var task = tasks.Where
22.             (x => x.Id == taskId).FirstOrDefault();
23.         if (task is not null) tasks.Remove(task);
24.     }
25. }

```

ASP.NET Core layer

In the ASP.NET Core layer, we define controllers that handle incoming HTTP requests, if any, and delegate the execution of business logic to the application layer. We do all the jobs in the **Home.razor** page and two other pages to manage the tasks: **Create.razor** to create a new task and an **Edit.razor** to edit a task.

The last action we have to take is the dependency injection. In the **Program.cs**, we have to add the following two lines of code:

```
1. builder.Services.AddScoped<ITaskPort, TaskAdapter>();
```

```
2. builder.Services.AddScoped<ITaskManagementPort,  
    TaskManagementAdapter>();
```

The code can be accessed from the github link.

By running our project, we will be able to see the following page:

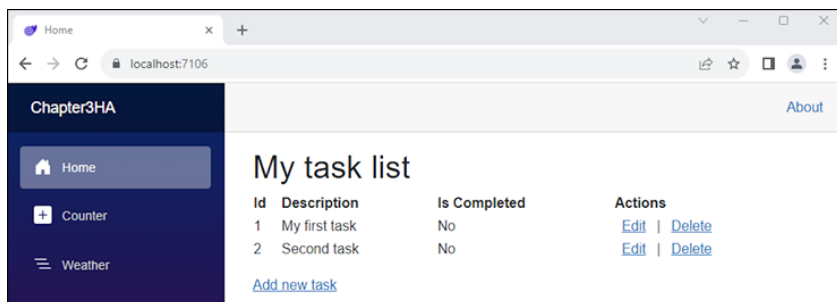


Figure 3.13: Task list with Hexagonal Architecture

As you might have observed, the data flow originates from the UI and extends toward the more tangible layers of the system.

In summary, our example project's data flow shows user interactions starting at the interface level, moving through application and domain layers, and reaching physical components. In this chapter, we implemented HA in an ASP.NET Core application, promoting modularity, testability, and maintainability by organizing the project into distinct layers.

Conclusion

In this chapter, we explored key architectural paradigms: MVC, Clean Architecture, DDD, and HA. Each emphasizes the separation of concerns, maintainability, and scalability. These paradigms enhance modularity, flexibility, and testability, especially in ASP.NET Core. This comprehensive exploration equips us to make informed decisions when designing modern software systems.

In the next chapter, we will see how RESTful APIs work in order to correctly structure the entry points of our application.

Points to remember

- **Foundational principles:** Understanding Clean Architecture,

DDD, MVC, and HA is crucial for designing robust software. These paradigms emphasize separation of concerns, dependency inversion, and Domain-Driven Design.

- **Effective framework utilization:** In ASP.NET Core, using the framework's features, such as dependency injection, is vital for clean, modular code, enhancing flexibility, scalability, and alignment with business needs.
- **Changing approach:** Applying architectural patterns and understanding business needs, as DDD suggests, is essential for developing dynamic, business-focused functionalities.

Multiple choice questions

1. **What is the primary focus of Clean Architecture?**
 - a. Optimizing performance
 - b. Separation of concerns and dependency inversion
 - c. Maximizing code reuse
 - d. Minimizing code complexity
2. **Which architectural pattern promotes the creation of a rich domain model and ubiquitous language?**
 - a. Model-View-Controller
 - b. Hexagonal Architecture
 - c. Clean Architecture
 - d. Domain-Driven Design
3. **What is the primary responsibility of the controller in MVC architecture?**
 - a. Rendering HTML views
 - b. Handling user input and orchestrating interactions between the model and view
 - c. Managing data persistence
 - d. Defining the structure of the application's domain model
4. **In Hexagonal Architecture, what are the roles of ports and adapters?**

- a. Ports define external systems, while adapters handle internal business logic.
- b. Adapters define external systems, while ports handle internal business logic.
- c. Both ports and adapters define external systems.
- d. Both ports and adapters handle internal business logic.

Answer key

Key terms

- **Model-View-Controller architecture:** Organizes applications into three components (Model, View, Controller) to separate concerns and facilitate maintainability in user interface design.
- **Clean Architecture:** Emphasizes modular design and clear boundaries between layers for maintainable and testable code.
- **Domain-Driven Design:** Focuses on understanding the business domain and creating a domain model that reflects it accurately.
- **Hexagonal Architecture:** Separates core business logic from external concerns using ports and adapters for flexibility and testability.

1 Image source: <https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 4

Designing RESTful APIs

Introduction

Today's web development requires robust **application programming interfaces (APIs)**. **REpresentational State Transfer (REST)** is the predominant architectural style for creating such APIs, offering key principles for network applications. This chapter explores the concepts of REST and RESTful APIs, analyzing their practical implementation. We examine the integration of minimal APIs into the **Model-View-Controller (MVC)** pattern in ASP.NET Core, evaluating its advantages and disadvantages. Minimal APIs offer lightweight structure and simplified syntax, making endpoint development and deployment easier. Through an application example, we illustrate best practices for creating robust and scalable solutions. Best practices in designing RESTful APIs are also discussed, highlighting the importance of maintainability and scalability.

Structure

In this chapter, we will cover the following topics:

- Concepts of RESTful APIs
- ASP.NET Core Web API
- Minimal APIs and MVC in ASP.NET Core
- Minimal API structure
- Best practices in designing RESTful APIs

Objectives

In this chapter, our main goal is to better understand how **HyperText Transfer Protocol (HTTP)** works to best use it in communication between two endpoints using REST principles and RESTful APIs. By the end of this chapter, we will have a comprehensive understanding of the HTTP protocol and its fundamental role in web communication. In particular, communication between client and server will be included.

Concepts of RESTful APIs

In this paragraph, we discover the foundational concepts of REST and RESTful APIs. REST, has emerged as a predominant architectural style for designing networked applications.

REST and RESTful

The terms **REST** and **RESTful** are closely related but have slightly different meanings. Following is the explanation:

REST is an architectural style for designing networked applications, introduced by *Roy Fielding* in his doctoral dissertation in 2000. It defines a set of principles for creating scalable, stateless, and loosely coupled distributed systems. These principles include statelessness, client-server architecture, uniform interface, resource-based interactions and **Hypermedia as the Engine of Application State (HATEOAS)**. REST focuses on the design of the system as a whole, emphasizing the separation of concerns and the use of standard protocols and conventions.

The term **RESTful** is an adjective used to describe systems, services, or APIs that adhere to the principles of REST. A system or API is

considered RESTful, if it follows the constraints and guidelines outlined by REST.

A RESTful API is designed by following the key points, as follows:

- In accordance with REST principles
- Featuring stateless communication
- Resource-based endpoints
- Standard HTTP methods
- Hypermedia-driven navigation

While **REST** refers to the architectural style itself, **RESTful** describes systems or APIs that conform to **REST** principles. A system can be **RESTful** if it implements the principles of **REST** effectively in its design and implementation.

Understanding REST Principles

As we said, **REST** is an architectural style for designing networked applications. At the core of REST are several principles that guide the design and implementation of APIs. Understanding these principles is essential for crafting efficient, scalable, and maintainable APIs. Let us delve into each principle to grasp its meaning.

Statelessness

REST advocates for stateless communication between clients and servers. This means that each request from a client to the server must contain all the information necessary to understand and process the request. The server does not store any client state between requests, which enhances scalability, reliability, and simplicity of the system. By decoupling requests, statelessness allows servers to treat each request independently, making it easier to scale horizontally and handle a larger number of clients.

Client-Server architecture

The Client-Server architecture principle in REST emphasizes the separation of concerns between the client and server components. The client is responsible for presenting the user interface and initiating requests, while the server handles processing those

requests, executing actions, and returning responses. This separation allows for scalability, flexibility, and modifiability, and applications can change from their side without impacting on the other side. For example, if we change the implementation of server the client is not forced to change. The only constraint is that the interface between client and server must not change.

Uniform interface

As we said before, uniform interface is the fundamental aspect of **REST** that simplifies the architecture and enhances the visibility of interactions. Following are the four key constraints:

- **Resource identification: Uniform Resource Identifiers (URIs)** serve as the means to identify resources. URI typically consists of a scheme (e.g., **http** or **https**), a host (e.g., **myexample.com**), a path (e.g. **/products/food**), and optionally query parameters or an identifier. With a well-defined URI structure, clients can easily access to API resources then contents of the server.

For example, consider an application where information about specific product can be retrieved by its identifier from clients. The URI for accessing a user's profile might resemble the following:

```
1. https://api.myexample.com/products/food/{product\_id}
```

In this URI, **{product_id}** serves as a placeholder for the identifier of the product. By substituting the placeholder with a real product identifier, clients can retrieve the complete information details for the required product.

- **Resource manipulation through representations:** In software systems, resources are interacted with via representations, which commonly take the form of JSON or XML documents. When clients wish to manipulate these resources, they send requests to servers, specifying the desired actions. Servers, in turn, respond with representations of the current state of the requested resource. By utilizing this approach, the coupling between clients and servers is minimized, as clients need not be aware of the internal data structures of the server. This

decoupling foster flexibility within the system architecture, enabling easier maintenance, scalability, and the potential for future changes without disrupting the overall functionality.

- **Self-descriptive messages:** Every message passed between clients and servers carries comprehensive metadata within itself. This metadata encapsulates essential details like content type, content length, caching directives, and other pertinent information necessary for the understanding and processing of the message.

In practical terms, this means that each message is self-sufficient and does not require additional external context or prior knowledge for interpretation.

To better understand these key concepts, we can see a concrete example, as follows:

Let us say a client sends a request to a server to get details about a product and the client-side application prepare properly the request. The message sent by the client might look as follows:

1. **GET** /products/food/**123** HTTP/**1.1**
2. **Host:** myexample.com
3. **Accept:** application/json

In this example, we can see some key points, as follows:

- The HTTP protocol version specified is 1.1.
- The client uses HTTP-GET method to retrieve information about the product with the identifier 123.
- The path **/products/food/123** specifies that we want a product in the category below called food.
- The **Accept** header tells the server that the client accepts responses in JSON format.

The server will respond with a message containing the requested information, which might be structured, as follows:

1. HTTP/**1.1** **200** OK
2. Content-Type: application/json
3. Content-Length: **120**

```
4. Cache-Control: max-age=3600
5. {
6.   "id": 123,
7.   "name": "Sample food product",
8.   "price": 9.99,
9.   "description": "This is a sample food product."
10. }
```

In this response, we can see two different parts: a header represented by the lines from 1 to 4 and a body represented by the lines from 5 to 10. We can also consider the following:

- In the header part, as follows:
 - o The **HTTP** status code **200 OK** indicates that the request was processed correctly.
 - o The **Content-Type** specifies that the response content is in JSON format.
 - o The **Content-Length** indicates the length of the response content expressed in number of char.
 - o the **Cache-Control** specifies caching directives for the response.
- The body of the response contains the requested product information, structured as a representation of a JSON object.

By embodying all relevant information within the message itself, self-descriptive messages bolster visibility and interoperability within distributed systems. Clients and servers can effectively communicate and interact without relying on external documentation or predefined conventions. This enhances the system's flexibility, as changes can be made without disrupting the overall communication flow.

- **Hypermedia as the engine of application state:** HATEOAS allows navigation between resources by embedding hyperlinks within representations. Clients then dynamically explore and traverse resources by following these hyperlinks provided by the server.

Let us consider a simple REST API for managing the usual product. In this case, we will be more specific, saying that the

products will be books in a library. When a client requests a book's details, the server responds with its data and links to related resources, like the author and similar books. We can figure out the request or response as follows:

o Request:

1. GET /books/123 HTTP/1.1
2. Host: myexample.com
3. Accept: application/json

o Response:

1. HTTP/1.1 200 OK
2. Content-Type: application/json
- 3.
4. {
5. "title": "La Divina Commedia",
6. "author": {
7. "name": "Dante Alighieri",
8. "link": "https://myexample.com/authors/alighieri"
9. },
10. "genre": "Classic",
11. "links": {
12. "self": "https://example.com/books/123",
13. "similar_books": "https://myexample.com/books?genre = Classic",
14. "reviews": "https://myexample.com/books/123/reviews"
15. }
16. }

In this response:

- The book *La Divina Commedia* is represented with its title.
- Information about the author, *Dante Alighieri*, is provided along with a hyperlink to retrieve more details about the author.

Hyperlinks to related resources are included under the **links** object, as follows:

- The **self-link** points to the resource representing the book itself.
- The **similar_books** link directs the client to other books in the same genre.
- The **reviews** link leads to the reviews for this book.

Including hyperlinks lets clients navigate the API dynamically, exploring related resources and interacting flexibly without prior knowledge of URL structures or endpoints.

In this way, Clients becomes more autonomous in navigating the API, reducing the need for tightly coupled implementations. Overall, HATEOAS empowers clients to interact with APIs in a dynamic and intuitive manner, enhancing the overall flexibility and evolvability of the system.

HTTP methods and status codes

HTTP request methods (also knowns as verbs) are standardized methods used by clients to indicate the desired action to be performed on a resource located on a server. Each method has a specific purpose and semantics, allowing clients to interact with resources in various ways. For each request from clients, servers will respond to these request methods via well-classified codes.

HTTP methods

In the following list, we will see the most common methods allowed from REST:

- **GET:** This method is used to retrieve data from a specified resource. It is a safe and idempotent operation, meaning it should have no side effects on the server and can be repeated without changing the state of the server and the client should get always the same response from the server.
- **POST:** This method is used to send data to be processed by a specified resource. It often involves creating a new resource on the server or updating an existing resource. Unlike GET, POST requests can have effects on the server and are not idempotent.
- **PUT:** This method is used to update or replace the entire

content of a resource with the data provided in the request. It is idempotent, meaning multiple identical requests should have the same effect.

- **DELETE:** This method is used to remove a specified resource from the server. It is also idempotent, meaning multiple identical requests should have the same effect.
- **PATCH:** This method is used to apply partial modifications to a resource. It is typically used when a client wants to update only specific fields of a resource without replacing the entire content. Like PUT, it is idempotent if the same patch request is applied multiple times.
- **HEAD:** This method is similar to GET, but it requests only the headers of the response without the actual body content. It is often used to retrieve metadata about a resource, such as its size or last modification date.
- **OPTIONS:** This method is used to retrieve the HTTP methods supported by a resource or the server's capabilities. It allows clients to determine which methods and headers are allowed when interacting with a resource.

In response to a request, the server provides a response accompanied by a status code. Particular emphasis is placed on the word **idempotent**, a concept derived from mathematics that can be summarized as follows:

Note: An idempotent operation produces the same result regardless of the number of times it is applied.

HTTP status codes are three-digit numbers returned by a server in response to an HTTP request from a client. They provide information about the result of the request and guide the client on how to proceed.

Status codes

HTTP status codes are numeric responses (three digits) from web servers that indicate whether a request was successful, encountered an error, or requires further action. They are classified in five categories, as follows:

- **1xx – Informational:** This category of status codes indicates

that the request has been received and is being processed. They are informative that we do not typically see in everyday web browsing.

- **2xx – Success:** This category of status codes indicates that the request was successfully received, understood, and accepted by the server. The most frequently used is 200 which means that the request is accepted and adequately processed.
- **3xx – Redirection:** This category of status codes indicates that further action is required to complete the request. They often indicate that the client needs to take additional steps, such as following a redirect URL.
- **4xx – Client error:** This category of status codes indicates that a client-side error occurred, such as invalid request syntax or unauthorized access, or simply that the resource being searched for was not found. In the latter case the very famous response is the 404-status code indicating that the requested resource was not found.
- **5xx – Server error:** This category of status codes indicates that a server-side error occurred while processing the request, such as server overload or internal server errors.

Correctly understanding HTTP methods and status codes is essential for building and consuming web services, because they determine how clients interact with servers and interpret server responses.

Following are some examples of common HTTP status codes:

- **200 OK:** The request was successfully processed.
- **404 Not Found:** The requested resource was not found on the server.
- **401 Unauthorized:** The request requires user authentication.
- **500 Internal Server Error:** A generic server error occurred while processing the request.

For more information about HTTP semantics, refer to the following official link:

<https://www.rfc-editor.org/rfc/rfc9110.html>

ASP.NET Core Web API

ASP.NET Core supports two modalities to build Web API, using controllers or using minimal API. In this paragraph, we will make an overview of both.

Endpoints and controllers in ASP.NET Core

In ASP.NET Core, when aiming to build a RESTful API endpoint, the conventional approach is to create a controller. A controller is a C# class that encapsulates the logic for handling incoming HTTP requests and generating corresponding outgoing responses. Controllers serve as the central entry point for routing and processing requests in ASP.NET Core applications. Each controller typically represents a logical grouping of related endpoints or actions that correspond to specific URI paths and HTTP methods. For example, the order management of a bookstore should generally have a single controller.

When we create a controller in ASP.NET Core for building RESTful APIs, we define action methods within the controller class. These actions will directly correspond to HTTP methods (like **GET**, **POST**, **PUT** and **DELETE**). They are responsible for processing incoming requests from clients and returning appropriate responses. The association between the action method and the HTTP method is done by decorating the action methods with specific attributes for each HTTP verb. For example, we will have attributes like **[HttpGet]**, **[HttpPost]**, **[HttpPut]**, **[HttpDelete]** and so on.

For instance, to create an endpoint for retrieving a list of resources for example, our task list seen in [Chapter 3, Architecture and Core Components](#), you would typically define a **GET** action method within the controller. Similarly, to create an endpoint for adding a new resource, you would define a **POST** action method.

We can take one of the examples from [Chapter 3, Architecture and Core Components](#) to use as a basis for reasoning. For example, we could think of having our application that instead of saving tasks in a list in memory, calls an external service via a REST API. This application will expose endpoints for all necessary methods of CRUD. For example, we could create an application from Visual Studio from the **ASP.NET Core Web API** template which we will

call **Chapter4APIControllers**.

The code can be accessed from the github link.

Visual Studio will provide us with a pre-packaged structure containing a sample controller. Since we will not need it, we will delete the existing controller and create another one from scratch called **TaskListController.cs** and always place it under the **Controllers** folder.

We will also need the data model that we used in the previous example and which for brevity of discussion we will limit ourselves to copying into our project but we will see in [Chapter 7, Architectural Principles](#) which architectural choice we can make to avoid code duplication.

So, as we said, our controller will be a classic C# class but with a few more attributes. Following is an example:

```
1. namespace Chapter4APIControllers.Controllers;
2. [Route("api/[controller]")]
3. [ApiController]
4. public class TaskListController : ControllerBase
5. {
6.     private readonly ITaskBridge tasksBridge;
7.     public TaskListController(ITodoBridge tasksBridge) = >
8.         this.tasksBridge = tasksBridge;
9.     [HttpGet]
10.    public IActionResult Get()
11.    {
12.        var tasks = tasksBridge.GetAllTasks();
13.        return Ok(tasks);
14.    }
15.    [HttpPost]
16.    public IActionResult CreateTodo([FromBody] TaskEntity
    ctedTask)
17.    {
18.        tasksBridge.CreateTodo(ctedTask);
19.        return CreatedAtAction("Summary", ctedTask);
20.    }
21. }
```

As we can see, on the top of the class we have the following code:

1. `[Route("api/[controller]")]`
2. `[ApiController]`

Representing the two attributes that will start to transform a simple C# class in a controller (in section *ASP.NET Core minimal API* we explain better how attributes works). Furthermore, the class will inherit from **ControllerBase** that put in practice the transformation in a real controller correctly building everything you need to handle HTTP requests and generate responses avoiding the most part of boilerplate code for developers. It provides access to the HTTP request context, including headers, query parameters, request body, and route data. It assists in generating HTTP responses by providing various helper methods for creating action results. These action results represent the outcome of processing a request and include options such as **OkResult**, **BadRequestResult**, **NotFoundResult**, **JsonResult**, **CreatedAtActionResult**, etc. As in our example we can see the **OkResult**.

ASP.NET Core controller-based API

Controllers are essential components that handle incoming HTTP requests and generate appropriate responses. They inherit from **ControllerBase** or **Controller**, providing a foundation for creating web APIs and MVC applications. Controllers leverage attributes to define routes, actions, and behaviors, enabling clean and organized request handling. Binding source parameter inference simplifies the process by automatically binding action parameters from various sources like query strings, route data, and request bodies. Additionally, ASP.NET Core's **ProblemDetails** class offers a standardized way to return detailed error information and status codes, enhancing API responses with consistent and informative error handling. This approach streamlines development and ensures robust, user-friendly web applications.

Controller and ControllerBase classes

A controller-based Web API is a layer that includes one or more class controllers. These class controllers inherit from the **Controller** or **ControllerBase** classes which belong to the ASP.NET Core framework. One of the most important functions of **ControllerBase**

is to provide the aids in routing HTTP requests to the appropriate action methods within controllers. It supports convention-based routing, attribute-based routing, or a combination of both, allowing developers to define the routing configuration that best suits their application's needs (further explained in *Chapter 5, Implementing Routing in ASP.NET Core*). The distinction between **Controller** and **ControllerBase** is that the **Controller** class derives from **ControllerBase**. The **Controller** class and its derived classes also support Web pages.

ControllerBase facilitates model binding, a process whereby data from the HTTP request (such as form data, query parameters, or JSON payloads) is automatically mapped to parameters of action methods or model properties. This simplifies the extraction and utilization of data from incoming requests within controller actions.

In the **ControllerBase** class, we have many methods and properties available to manage HTTP requests. For instance, if we have to manage a CRUD based application, we have a method named **CreatedAtAction** available, as we can see in the following example:

```
1. [HttpPost]
2. public ActionResult<Book> Create(Book book)
3. {
4.     return CreatedAtAction(nameof(GetById),
5.         new { id = book.Id }, book);
6. }
```

The **CreatedAtAction** method, automatically respond as **201 Created** if method success, but we can manage the response status code adding attributes on the method signature as follows:

```
1. [HttpPost]
2. [ProducesResponseType(StatusCodes.Status201Created)]
3. [ProducesResponseType(StatusCodes.Status400BadRequest)]
4. public ActionResult<Book> Create(Book book)
5. { // OMISSIS
```

In this specific case, we return **201 Created** or **400 BadRequest** according to the result of the method action called. The **StatusCodes** enum wraps as constants the most used status codes.

Attributes

Attributes are special methods used to configure behavior of web API and are included in **Microsoft.AspNetCore.Mvc** namespace. In the previous example, we used three of them. In line 1, we can find the verb HTTP to which the method will answer. In this case, it is **POST**. The other two lines are attributes of response as seen before.

We can have some other attributes we can use profitably. Following is a list of some of them:

- **[ApiController]**: This attribute is applied to the class, indicating that the underlying class is a controller. The use of this attribute automates model binding management and 400 error responses with **ProblemDetails** (for model validation), and it requires the use of attribute routing.
- **[Route]**: It is one of the most used. It specifies URL pattern for the correspondent controller or action. The use of this attribute is simple and the route attribute should be explicit as follows: **[Route("[controller]")]** or **[Route("api/[controller]")]**. This attribute is used on the controller class as the previous one.
- **[HttpGet]**: As seen before with **[HttpPost]**, this attribute identifies that the correspondent action maps to the HTTP GET as verb.
- **[Consumes]**: This special attribute specifies which data types are accepted from the method action. The use is quite easy: **[Consumes("application/json")]**.
- **[Produces]**: In a similar way as **[Consumes]** attribute, it specifies data types that an action return. The use is quite similar to the previous attribute: **[Produces("application/json")]**.

Binding source parameter inference

In Web API methods, parameters are often passed to perform the action correctly. For example, integer parameters such as an id or even an object parameter such as the instance of a book class are passed. If a parameter is passed directly in the route, you do not need to declare its provenance but if it comes from another part of the HTTP message, then you need to specify its provenance so that you can correctly retrieve the parameter from the incoming HTTP

message. The most used attributes are as follows:

- **[FromBody]**: Indicates that the parameter is in the request body.
- **[FromForm]**: Indicates that the parameter is in the form data in the request body.
- **[FromHeader]**: Indicates that the parameter is in the request header.
- **[FromQuery]**: Indicates that the parameter is in the request query string parameter.
- **[FromRoute]**: Indicates that the parameter is in the route data from the current request.
- **[FromServices]**: Indicates that the parameter is in the request service injected as an action parameter.

A simple call in ASP.NET Core passing an integer parameter, is as follow:

```
1. var response = await httpClient.GetAsync($" /Books/{123}");
```

The correspondent action should be as follows:

```
1. [HttpGet(Name = "GetById")]
2. public ActionResult<Book> GetById(int id)
3.     => booksInMemoryStore.FirstOrDefault(p => p.Id == id)
   ?? new();
```

As we can see, the parameter **id** is passed directly as method parameter. An object parameter that needs to be retrieved from body (for instance) is as follows:

```
1. [HttpPost(Name = "CreateBook")]
2. public IActionResult CreateBook([FromBody] Book book) =>
   Ok(book);
```

Calling this action from a client may be slightly more complex, as follows:

```
1. string json = JsonSerializer.Serialize(myNewBook);
2. var content = new StringContent(json, Encoding.UTF8,
   "application/json");
3. var response = await httpClient.PostAsync(url, content);
```

As we can see, the **[FromBody]** attribute is assigned to the

controller action in order to indicate in which part of the request message we will find the sent object.

Problem details to return error status codes

ASP.NET Core transforms an error code in a response with an object **ProblemDetails** as body. If we decide to return a status code correspondent to an error code (greater than 400) the standard behavior is that the object **ProblemDetails** is automatically included in body response. There are two schools of thought around returning errors from a Web vAPI call, as follows:

- Return the proper status code and let ASP.NET Core include the standard **ProblemDetails** in body
- Always return a 200 Ok and include in the body a **ProblemDetails** object specific to the error encountered

It is hard to say what the best solution is, but we will certainly need to evaluate the entire application and be consistent throughout the Web API layer that will be exposed.

Let us consider this example:

```
1. [HttpGet]
2. public IActionResult Get()
3. {
4.     try
5.     {
6.         return Ok(GetItems());
7.     }
8.     catch (Exception ex)
9.     {
10.        return Ok(new ProblemDetails()
11.        {
12.            Status = 500, Detail = ex.Message
13.        });
14.    }
15. }
```

As we can see, even if an exception is raised, the response is always 200 but we craft properly a **ProblemDetails** object that represents the error raised. In this case, our response will be as follows:

```
1. HTTP/1.1 200 OK
2. Content-Type: application/problem+json; charset=utf-8
3. date: Fri,22 Mar 2024 23:00:52 GMT
4. server: Kestrel
5. timing-allow-origin: *
6. {
7.   "status": 500,
8.   "detail": "An exception has occurred in GetItems()."
9. }
```

Now, consider to return an error status code, a simple **NotFound** with the error message as return parameter. The example method should be as follows:

```
1. // OMISISS
2. catch (Exception ex)
3. {
4.   return NotFound(ex.Message);
5. }
```

Now, our response will be different, as follows:

```
1. HTTP/1.1 404 NOT-FOUND
2. Content-Type: text/plain; charset=utf-8
3. date: Fri,22 Mar 2024 23:00:52 GMT
4. server: Kestrel
5. timing-allow-origin: *
6. An exception has occurred in GetItems().
```

As we can see, the response body is the text message of our **ex.Message** passed back to the client request. No other information is required and no other details are attached. The response status code is quite clear and is not necessary to go deeper in the message body to understand why there is an error.

ASP.NET Core minimal API

Minimal APIs represent a lean paradigm within ASP.NET Core for creating efficient HTTP APIs. With this approach, developers can create fully functional REST endpoints with minimal code and configuration. By avoiding conventional construction practices, we can fluidly articulate API paths and actions, avoiding heavy

controllers and unnecessary, excessive code. This minimalist methodology promotes rapid development and improves the clarity and simplicity of API implementations.

Only to understand what we are speaking, take a look to the minimal API as follows:

```
1. app.MapGet("/", () => "Hello World!");
```

This simple **GET** API, returns a **Hello World!** String but this API in a controller method looks as follows:

```
1. [ApiController]
2. [Route("[controller]")]
3. public class SimpleController : ControllerBase
4. {
5.     [HttpGet]
6.     public IActionResult Get()
7.     {
8.         return Ok("Hello World!");
9.     }
10. }
```

As we have seen in the previous paragraph, in this API we have a lot of boilerplate code that in our case is not necessary. With minimal API we have one line of code and in a controller approach ten. Everything we have seen regarding the controllers still remains valid. The parameters, attributes and problem details response however remain valid for the minimal API.

Minimal APIs and MVC in ASP.NET Core

In the previous paragraph, we delved into the fundamental concepts of REST and RESTful API, exploring its principles and key concepts. Now, we dive into the practical aspect of building web API using minimal APIs in an ASP.NET MVC application.

Minimal APIs

As we have seen in the previous paragraph, minimal APIs presents simplified methods for building web APIs with minimal overhead. Unlike traditional controller and routing configurations, minimal APIs allows delineate endpoints using concise syntax, thus

improving the cleanliness and maintainability of the code base. Let us start creating a simple minimal API to handle **Create, Read, Update, Delete (CRUD)** operations for the **TaskList** application mentioned in previous paragraph. Starting simply, we can open the **Program.cs** file in our project, as follows:

```
1. var app = builder.Build();
```

Just after the line, we can put the following code:

```
1. app.MapGet("/tasks", () => { // Retrieve and return list of tasks
    }
2. app.MapGet("/tasks/{id}", (int id) =>
3.     { // Retrieve and return task with specified id });
4. app.MapPost("/tasks", (TaskEntity todo) => { // Create new
    task });
5. app.MapPut("/tasks/{id}", (int id, TaskEntity todo) =>
6.     { // Update task with specified id });
7. app.MapDelete("/tasks/{id}", (int id) =>
8.     { // Delete task with specified id });
```

In this code snippet, we define endpoints for fetching all tasks in the list, fetching a single task by its ID, creating a new task, updating an existing task, and deleting a task. The syntax is simple to understand, promoting faster development and reduced boilerplate code.

Minimal API with MVC

While minimal APIs excel in simplicity and brevity, MVC remains a robust pattern for building complex web applications with good and clear separation of concerns. MVC divides an application into three interconnected components, that is, Model, View, and Controller, facilitating maintainability and testability. In [Chapter 3, Architecture and Core Components](#), we have seen how roughly treat the MVC pattern and architecture. In this case we will see how this concept will be extended as well separating the Models, the Views and extending the concept of Controllers in a different way. It is the MVC pattern but in a different and modern conception. Let us extend our TaskList application using MVC to provide a more structured approach.

Model

As seen in previous paragraph, the model we can use should be the **TaskEntity** to define the common model between layers. To well separate the concerns and to use the same model from the server and client, we can add to our solution a new project that we can name **Chapter4 MinimalAPIWithMVC.Shared** to be distinguish from the others but with the same namespace root, as follows:

```
1. namespace Chapter4APIControllers.Shared.Models;
2. public class TaskEntity
3. {
4.     public int Id { get; set; }
5.     public string Description { get; set; } = string.Empty;
6.     public bool IsCompleted { get; set; }
7. }
```

Controller

As we said at the beginning of this paragraph, we replace the controllers with minimal APIs. Instead of return back a view, as a standard Controller, the Web API returns only data and let the Blazor runtime manage the render of the views according to the route's definitions. Using the Blazor Web App template, we can figure out the scenario where the web API serves only to retrieve data from server. In our case, we have to modify a little bit the standard template of Blazor Web App because we want a proper distinguish of front end and backend. In our case, the front end is interpreted by the **.Client** project while the backend is interpreted by the **Chapter4MinimalAPIWithMVC** project. For the first thing, we have to move into the **.Client** all the pages and the layouts (in the **Chapter4MinimalAPIWithMVC** project the Layout folder and Page folder are under Components folder but do not move the **Error.razor** page). Set the **@rendermode** properly in each page but not in the **MainLayout**. Fix the namespace error in **Routes.razor** and that is it. Now our main project named **Chapter4MinimalAPIWithMVC** can host minimal API that provides responses to the front-end requests. Let us improve a little bit our minimal API in **Program.cs** as follows:

```
1. // Minimal API
```

```

2. app.MapGet("/tasks", (ITaskRepository repo) => repo?.GetAll());
3. app.MapGet("/tasks/{id}", (int id, ITaskRepository repo)
4.     => repo?.GetTaskById(id);
5. app.MapPost("/tasks", ([FromBody] TaskEntity newTask,
ITaskRepository repo)
6.     => repo?.CreateNewTask(newTask);
7. app.MapPut("/tasks", ([FromBody] TaskEntity newTask,
ITaskRepository repo)
8.     => repo?.UpTask(newTask);
9. app.MapDelete("/tasks/{id}", (int id, ITaskRepository repo)
10.    => repo?.DeleteTaskById(id);

```

In the minimal API, we can see the injection of a repository (for details about repository, refer to [Chapter 3, Architectures and Core Components](#) the paragraph **Repositories and Repository Pattern**) able to elaborate properly a request. In our **TaskList** application, we need a CRUD elaboration. Our repository will be a simple class with CRUD with four methods that act on a simple list as a data archive. This class has a correspondent interface through which we can build with the dependency injection. In the **Program.cs**, now we can craft the dependency injection of the class as follows:

```

1. builder.Services.AddSingleton<ITaskRepository,
TaskRepository>();

```

View

The views of our application are as follows:

- Home page
- Create page
- Edit page

From each page we will call the API to retrieve data or send data to backend. For this reason, we need to build the HTTP Client able to create calls to the Web API then to our Minimal API endpoints. In the main project **Chapter4MinimalAPIWithMVC**, in **Program.cs** file, we have to build it as follows:

```

1. builder.Services.AddScoped(sp =>
2.     new HttpClient
3.     {

```

4. `BaseAddress = new`
5. `Uri(builder.Configuration["baseAddress"]`
6. `?? "http://localhost:5052")`
6. `});`
7. `builder.Services.AddHttpClient();`

Using the configurations in **appsettings.json**, we can add the following key:

1. `"baseAddress": "https://localhost:7229",`

Be careful, the ports 5052 and 7229 are the port configured in my example project in the **launchSettings.json** under Properties folder. Replace these ports numbers with the ones set in your project. Refer to [Chapter 3, Architecture and Core Components](#) in the section, *Clean Architecture in practice with ASP.NET Core project*, able to show and manage the list of tasks. In those pages, we have calls to the back end and the repository through `HttpClient` as follows:

1. `var senderStatus = JsonSerializer.Serialize(taskEntity);`
2. `var requestContent = new StringContent(senderStatus,`
3. `Encoding.UTF8, MediaTypeNames.Application.Json);`
4. `await httpClient.PutAsync($" /tasks", requestContent);`

How we can see, the requests with verb **PUT** (or in a similar way with **POST**), we send data as **JSON** serialized object as `HttpContent` in the body of message and retrieved in our minimal API with the attribute `[FromBody]`. This hybrid approach allows us to capitalize on the simplicity of minimal APIs for certain tasks, while leveraging the comprehensive features of MVC for others, ultimately resulting in a flexible and scalable application architecture.

Minimal API structure

Using minimal APIs reveals a paradigm shift in ASP.NET Core development, introducing a more streamlined and intuitive approach to building Web APIs. Unlike traditional controllers and routing configurations, minimal APIs simplify the definition process of the endpoint using concise syntax, eliminating the need for extensive boilerplate code. This minimalist structure emphasizes simplicity and clarity, allowing developers to focus directly on defining the core functionality of their APIs without unnecessary

overhead. However, in a real scenario, the minimal APIs seen in the previous paragraph are difficult to maintain and evolutionary can be complicated and chaotic increasing complexity as shown in [Chapter 1, Introduction to ASP.NET Core](#). This complexity stems from the absence of a modular, scalable architecture. Poor separation of responsibilities and lack of isolation between components can cause small changes to impact the entire system unexpectedly. Without modularity, understanding interactions becomes harder, increasing errors and complicating troubleshooting. In practice, key measures can address these issues effectively.

First of all, we can move the minimal APIs in a more maintainable extension method but it is better to see in practice how these methods are implemented. The idea is to separate the API as seen in [Figure 4.1](#):

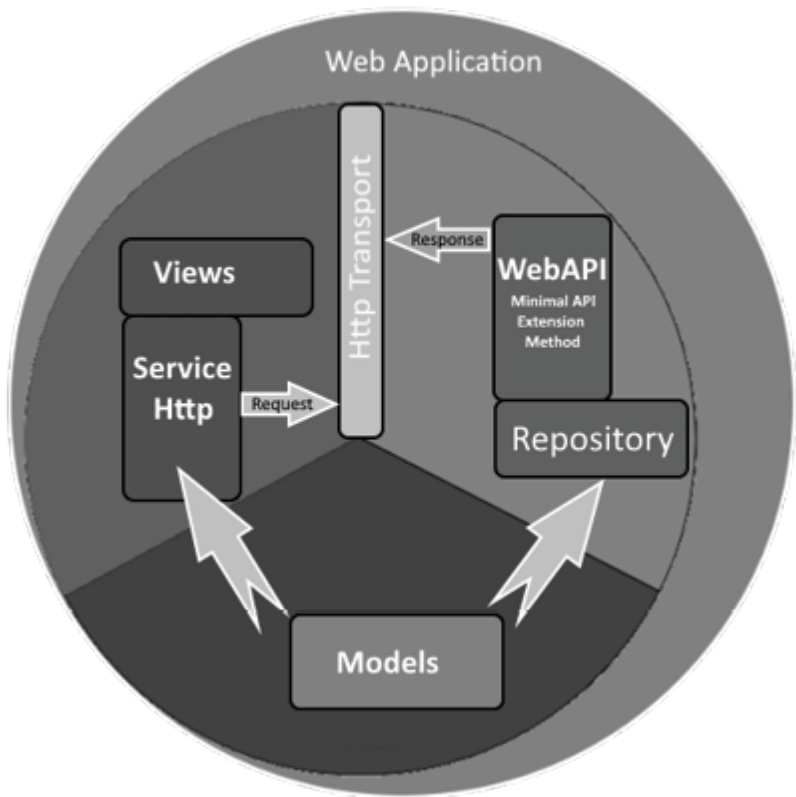


Figure 4.1: Minimal API application

If we take as example, the above minimal API can be seen as follows:

```
1. // Minimal API
2. app.MapGet("/tasks", (ITaskRepository repo) =>
3. {
4.     // Retrieve and return list of tasks
5.     return repo?.GetAll();
6. });
7. app.MapGet("/tasks/{id}", (int id, ITaskRepository repo) =>
8. {
9. // OMISSIS
```

According to [Figure 4.1](#), we can cut our application in different parts so that we can better manage and maintain the whole application according to the lifecycle. For the first approach, we can start to cut from the View parts. Only for example, we can take the method into the **Edit.razor** page able to update our task as follows:

```
1. private async Task SubmitForm()
2. {
3.     if (taskEntity.Description.Trim().Length > 0)
4.     {
5.         var senderStatus = JsonSerializer.Serialize(taskEntity);
6.         var requestContent = new StringContent(senderStatus,
7.             Encoding.UTF8, MediaTypeNames.Application.Json);
8.         await httpClient.PutAsync($"tasks", requestContent);
9.         NavManager.NavigateTo("/");
10.    }
11. }
```

As we can see, in this method, the *HttpClient* is directly used to send a request from the front-end to the Web API. During maintenance, it forces us to run after all the pages that contain HTTP calls. Most of times, we forget someone causing bugs or unexpected behavior.

Best practices in designing RESTful APIs

Generally speaking, designing a RESTful API involves more than just implementing endpoints and methods, it requires careful

consideration of various factors to ensure the API is intuitive, maintainable, and scalable. We will find the entire code example on [github link](#), while in this chapter, only the salient and important parts will be shown for the purpose of discussing the topic.

Client side

In this refactoring of our application, we will start from the **.Client** part. We will use a top-down approach, that is, an approach from top (front-end) to bottom (back-end). The first refactoring should impact the http calls to the Web API from the pages.

Services

By isolating data access logic in a service, Razor code is kept free from HTTP call details, making it more dynamic, agnostic, and focused solely on presentation. This approach reduces coupling between the user interface and the endpoints, simplifying future changes to API structures without affecting Razor code. Additionally, centralizing endpoint management in a single service streamlines maintenance, ensures consistent API usage, and minimizes the risk of errors or redundant implementations. The interface of the service is as follows:

```
1. public interface ITaskListService
2. {
3.     Task<bool> Create(HttpClient httpClient, TaskEntity
        taskEntity);
4. Task<List<TaskEntity>?> ReadAll(HttpClient httpClient);
5. Task<TaskEntity?> ReadById(HttpClient httpClient, int id);
6. Task<bool> Update(HttpClient httpClient, TaskEntity
        taskEntity);
7. Task<bool> Delete(HttpClient httpClient, int id);
8. }
```

There is another pain point we have to solve before implementing a class on this interface, the address of API endpoints. We must not write them as hardcoded string because we may have typos or copy or paste errors that will drive us crazy to look for non-existent errors without considering that the complexity of the system is

increased exponentially (refer to [Chapter 1, Introduction to ASP.NET Core](#)). We can manage this problem by writing them into a static class shared between front end and back end. In this approach, we still write the strings as hardcoded, but we will have a **single source of truth** and a single typo in the endpoint address would not cause any problems. This class in our example is simply named **Endpoints** and it will be physically in the **.Shared** project and in Models folder. In this case, we have only one context and one **Endpoints** static class, but if we will have more than a single context that is, in a DDD architecture, the best way is to split the **Endpoints** static class in a more specific class according to his bounded context. So, in our case, **Endpoints** static class will be as follows:

```
1. public static class Endpoints
2. {
3.     public const string TasksRootApiEndpoint = "api/chapter4";
4.     public const string ActualVersion = "v1";
5.     public const string TasksApiEndpoint = "tasks";
6. }
```

In our case, we can have a single address for the endpoint because we **play** with different http verbs. If we need some customization (like the id parameter), we can add it directly into the method we need it. The other two parts composes the whole address. The root endpoint is the address where all this group of API refers to. The **ActualVersion** is the versioning of the API. Now, we are ready for the implementation of **ITaskListService** that will be as follows:

```
1. public class TaskListService : ITaskListService
2. {
3.     public async Task<bool> Create(HttpClient httpClient,
        TaskEntity taskEntity)
4.     {
5.         var senderStatus = JsonSerializer.Serialize(taskEntity);
6.         var requestContent = new StringContent(senderStatus
7.             , Encoding.UTF8
8.             , MediaTypeNames.Application.Json);
9.         var response = await httpClient.PostAsync
10.             ($"{Endpoints.TasksRootApiEndpoint}
11.             /{Endpoints.ActualVersion}
```



```

12.         /{Endpoints.TasksApiEndpoint}", requestContent);
13.
14.         return response.IsSuccessStatusCode;
15.     }
16.     // OMISSIS

```

The service class will be managed from the dependency injection of ASP.NET Core and each method will be called from the pages as follows:

```

1.         builder.Services.AddScoped <ITaskListService,
TaskListService> ();

```

We need a scoped service because it is necessary from each client.

My personal suggestion is to manage the “/” which separates the three parts of the string that makes up the entire endpoint address directly in the composition of the address.

In this way, we will achieve the goal of separating the frontend code from Web API calls and a much cleaner frontend code. Furthermore, the service class will be much more maintainable than direct calls in the pages.

Server side

In the backend side, we can restructure the minimal API layer in a more efficient structure. Let us take the last implementation of our minimal API layer. The whole code was written in the **Program.cs** file. Try to figure out an application with hundred methods, the **Program.cs** will become suddenly a classic **big ball of mud** or **spaghetti code**. This is not our goal for maintainable code. So, let us discuss, how to restructure these methods to be modular and maintainable.

Extension method refactoring

For the first thing, we can create a folder under the main project named **ExtensionMethods**, then, we will create a static class. We will name the class **EndPointRouteBuilderExtension** and it will be able to extend our **Program.cs** class and in particular the **IEndpointBuilder** (the **app** variable in **Program.cs**) and host all our minimal API methods.

Following is how the class will look like:

```

1. public static class EndPointRouteBuilderExtension
2. {
3.     public static IEndpointRouteBuilder UseTaskListEndpoints(
4.         this IEndpointRouteBuilder app)
5.     {
6.         app.MapGroup(Endpoints.TasksRootApiEndpoint).MapApi();
7.         return app;
8.     }
9.     private static RouteGroupBuilder MapApi(this
10.        RouteGroupBuilder
11.        group)
12.    {
13.        _ = group.MapGet($"{Endpoints.ActualVersion}/tasks",
14.            GetAllTasksApi);
15.        _ = group.MapGet($"{Endpoints.ActualVersion}/tasks/
16.            {{id}}",
17.            GetTaskByIdApi);
18.        // OMISSIS
19.        return group;
20.    }
21.    private static IResult GetAllTasksApi(ITaskRepository repo)
22.        => Results.Ok(repo?.GetAll());
23.    private static IResult GetTaskByIdApi(int id, ITaskRepository
24.        repo)
25.        => Results.Ok(repo?.GetTaskById(id));
26.    // OMISSIS
27. }

```

In this part, we reuse the same class **Endpoints** defined in the previous paragraph and will use them in the extension method in the back-end too. The refactoring impacts in four main parts as follows:

- The first part is using the ASP.NET Core minimal API groups. Here we can see the use of group base address defined in **Endpoints** class and used here to craft the group.
- The second, is the **MapApi** extension method. It is able to map each method of group to a single private method so that we can properly manage any future modification.

- The third, is that we can manage properly the versioning of the API through the Endpoints class and in particular with **Endpoints.ActualVersion**.
- Fourth and last big change is the use of a repository class. In this way, we can manage separately the logic of the application.

The extension method should be declared and consumed in the **Program.cs** file as follows:

```
1. // Use Minimal API structured as extension method
```

```
2. app.UseTaskListEndpoints();
```

At this stage, our application is ready, incorporating the best practices outlined in this chapter. We can now apply these measures to enhance our real-world applications.

Conclusion

The chapter provided a comprehensive exploration of REST and RESTful APIs, highlighting their crucial role in modern web development. Our review provided a deeper understanding of REST principles and their impact on API design and implementation for seamless client-server communication. Exploring minimal APIs within the ASP.NET Core MVC model highlighted their lightweight nature, simplified syntax, and advantages for rapid development, while addressing modular architecture's importance. By discussing best practices, we identified tools and methodologies to design efficient, maintainable, and scalable minimal APIs that support version control and streamlined deployment. As Web development continues to evolve, the knowledge we gain in this chapter will help us create modern, robust, and efficient applications.

In the next chapter, we will explore routing in ASP.NET Core, focusing on conventional and attribute-based approaches, route templates, constraints, and query strings, while emphasizing best practices to create scalable, intuitive, and efficient navigation for modern web applications.

Points to remember

- **Fundamental principles:** Understand the core principles of

REST, including statelessness, uniform interface, and resource identification.

- **Minimal APIs and extension methods:** Appreciate the lightweight nature and simplified syntax of minimal APIs, enabling rapid development and deployment of API endpoints wrapped in a well separated and maintainable extension methods.
- **Enhancing efficiency:** Understand how minimal APIs can improve efficiency by improving maintainability, version control, and scalability, and be equipped with the knowledge and tools needed to design efficient minimal APIs.

Multiple choice questions

1. What is the HTTP method commonly used for retrieving data in RESTful APIs?
 - a. GET
 - b. POST
 - c. PUT
 - d. DELETE
2. Which characteristic is typically associated with minimal APIs?
 - a. Heavyweight and complex syntax
 - b. Slow development and deployment process
 - c. Lightweight nature and simplified syntax
 - d. Limited scalability and performance
3. Which of the following best describes an endpoint in the context of web development?
 - a. A specific URL where API requests are sent
 - b. The client-side code responsible for rendering web pages
 - c. A programming language commonly used for backend development

- d. The process of securing a web application against cyber threats
4. What is the most common HTTP response status code for a successful GET request?
- a. 200 OK
 - b. 400 Bad Request
 - c. 404 Not Found
 - d. 500 Internal Server Error

Answer key

Key terms

- **REpresentational State Transfer:** An architectural style for designing networked applications, emphasizing statelessness, uniform interface, and resource identification.
- **Application programming interface:** a set of rules and protocols that allows different software applications to communicate with each other. For extension **Minimal API** are lightweight APIs with simplified syntax, facilitating rapid development and deployment of API endpoints.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 5

Implementing Routing in ASP.NET Core

Introduction

Routing is a fundamental aspect of any web application, as it determines how incoming requests (from the user) are mapped to specific endpoints within the application, i.e., the Web API. In ASP.NET Core, routing is managed by a dedicated middleware (for more information about middleware, refer to [Chapter 6, Middleware and Extensibility](#)), which allows you to define routes and map them to controller actions or manage routes in minimal APIs.

This chapter will explore the complexities of implementing routing in ASP.NET Core, which orchestrates the flow of request processing in web applications. Routing is a critical component in ASP.NET Core and serves as the mechanism that maps HTTP requests inbound to the appropriate endpoints. This chapter aims to provide a comprehensive understanding of routing, from its fundamental concepts to advanced implementation techniques, giving readers the tools needed to design efficient, scalable, and maintainable web applications.

We will start by exploring the theoretical underpinnings of routing and trying to understand how routing works in an ASP.NET Core application. We will also study what endpoints are and how they work to manage routes correctly. We will also cover advanced topics such as custom routing constraints, manipulating routing data, and integrating middleware into the routing process.

We will also analyze a series of best practices and bad practices related to routing. We will see how to avoid the biggest mistakes and prevent problems resulting from incorrect use of routes.

Structure

In this chapter, we will cover the following topics:

- Introduction to routing in ASP.NET Core
- URL mapping to controllers and actions
- Configuring and customizing routing
- Route constraints and parameters
- Best practices and tips for routing

Objectives

By the end of this chapter, we will understand the fundamentals of routing in ASP.NET Core using controllers and minimal API. In addition, routing should be configured for our meaning and needs. Then, we will learn how to configure routing using both convention-based and attribute-based approaches. We will explore the concept of route constraints and parameters and their role in defining precise routing configurations. We will gain insights into best practices and tips for optimizing routing configurations in ASP.NET Core applications.

Introduction to routing in ASP.NET Core

In ASP.NET Core, routing is a key feature that maps incoming HTTP requests to specific controller actions. It enables developers to define URL patterns, facilitating clean and manageable URLs. With support for attribute-based and convention-based routing, ASP.NET Core provides flexibility and control over request handling and

navigation.

Understanding routing

Routing is the process of matching an HTTP request with a method that can respond to a well-defined path called an **endpoint**.

Endpoints are the units of the application containing the executable code to handle requests. They are defined in the application and configured during application startup. The endpoint matching process can extract values from the request URL and provide those values for request processing. By using endpoint information from the application, routing can also generate URLs that map to the endpoints themselves.

There are several routing concepts available in applications. Routes can be based on MVC controllers, Razor Pages, minimal APIs, SignalR, or gRPC. In this chapter, we will focus solely on routing based on controllers and minimal APIs.

One of the routing concepts on which the ASP.NET Core routing system is based on is the concept of **route templates**. It defines the pattern of incoming URLs and how they should be matched to specific actions in your application. Route templates consist of placeholders for route parameters, which are variables extracted from the URL.

Let us consider the following base example of a convention-based routing:

1. `app.MapControllerRoute(`
2. `name: "default",`
3. `pattern: "{controller = Home}/{action = Index}/{id?}");`

In this example, we are using the **MapControllerRoute** method to define a route named **default**. The route template **{controller = Home}/{action = Index}/{id?}** specifies that incoming requests should be mapped to controller actions based on the controller and action names provided in the URL. The **{id?}** segment indicates that the **id** parameter is optional.

Routing uses a pair of middleware components registered through **UseRouting** and **UseEndpoints**, which are explained, as follows:

- **UseRouting** integrates route matching into the middleware

pipeline, examining the defined endpoints within the application and identifying the most suitable match based on the incoming request.

- **UseEndpoints** incorporates endpoint execution into the middleware pipeline, executing the associated delegate linked with the chosen endpoint.

Typically, no application should call **UseRouting** directly. They are already configured, which sets up a middleware pipeline (see [Chapter 6, Middleware and Extensibility](#), for more details) that encapsulates the middleware added in **Program.cs**. At times, for some reason specific to the application domain, applications need to adjust the execution sequence of **UseRouting** by explicitly invoking these methods. In this case, the sequence of **UseRouting** changes according to the needs of the business and application domain.

The following is an example of how to call directly the **UseRouting** middleware:

```
1. var builder = WebApplication.CreateBuilder(args);
2. var app = builder.Build();
3.
4. /* CUSTOM MIDDLEWARES */
5.
6. app.UseRouting();
7.
8. /* OTHER MIDDLEWARES */
9.
10. app.MapGet("/", () => "Hello World!");
11. app.Run();
```

When a custom middleware is called before **UseRouting**, it is a pattern explicitly intended by the application, which, in this case, must initialize a middleware before route evaluations. Finally, registration with the **MapGet** method defines an endpoint using the minimal API approach to retrieve the desired string after the middleware **UseRouting** route matches.

Understanding endpoints

An endpoint refers to a specific location within a web application identified through a specific and strong combination of URL and

HTTP methods. The corresponding endpoint is triggered to execute the associated code when a request is made to a specific URL using a particular HTTP verb. Endpoints serve as entry points for interaction with the application, allowing users or other systems to perform specific actions or retrieve desired information via structured requests.

To configure endpoints for the applications to use it, ASP.NET Core allows us a specific method (as said before) called **UseEndpoints**. Similar to the **UseRouting** method, even **UseEndpoint** should not be used directly by our applications.

A simple endpoint using the principles of minimal API approach is as follows:

```
1. app.MapGet("/", () => "Welcome in Chapter 5!");
```

The standard output will be as follows:



Figure 5.1: Minimal API output

As we can see, the output is the flat returned string.

In this case, we are using the minimal API in **Program.cs**. We have limited control over the endpoint, and as we have seen in [Chapter 4, Designing RESTful APIs](#), we can properly configure an extension method containing minimal API.

The second mode to properly build an endpoint useful for API is to use **UseEndpoints** middleware as follows:

```
1. // Route configurations
2. app.MapGet("/", async context =>
3. {
4.     await context.Response.WriteAsync("Welcome to Chapter 5!");
5. });
6. app.MapGet("/about", async context =>
7. {
8.     await context.Response.WriteAsync("About chapter 5.");
9. });
```

In this use example, we will have a warning around the **UseEndpoints** saying, *ASP0014: Suggest using top level route registrations*, covertly hiding the suggestion to avoid using **UseEndpoints** in favor of top-level route registration we saw previously. However, if it is genuinely necessary, do not hesitate to utilize it. If the warning bothers you, you can avoid it by using **#pragma warning disable ASP0014** before and after the code block.

While **MapGet** offers fine-grained control over individual types of HTTP requests for specific routes, **UseEndpoints** provides greater flexibility in configuring routes. It allows you to handle complex scenarios more efficiently and dynamically adapt application behavior.

Routing in minimal APIs

A small emphasis will be placed on the routing system in the minimal API. We can have a simplified routing system to avoid many addresses hardcoded as strings in our code that are difficult to maintain. For reference, see [Chapter 4, Designing RESTful APIs](#), in the refactoring section.

Therefore, API minimal routing is based on route groups and obtained using the **MapGroup** method, as follows:

1. `app.MapGroup("/public/tasklist")`
2. `.MapTaskListApi();`

The **MapTaskListApi()** is the name of an extension method that groups all the minimal APIs belonging to the same group. For instance, we can have the method **MapTaskListApi()** as follows:

1. `public static RouteGroupBuilder MapTaskListApi(this RouteGroupBuilder group)`
2. `{`
3. `group.MapGet("/", GetAll);`
4. `group.MapGet("/{id}", GetById);`
5. `group.MapPost("/", Create);`
6. `group.MapPut("/{id}", Update);`
7. `group.MapDelete("/{id}", Delete);`
8.
9. `return group;`
10. `}`

Each method refers to the **group** defined before. As we saw in the previous chapter, the strings containing the group endpoint should be written in a separate static class to better refactor it. In this way, we can properly manage the route in minimal APIs.

URL mapping to controllers and actions

In ASP.NET Core, URL mapping plays a pivotal role in directing incoming requests to the appropriate controller actions or minimal API. By understanding how URLs are mapped to controllers and actions, we gain the ability to design intuitive and efficient routing structures.

Route parameters

Route parameters allow the developer to extract values from the URL and use them within your controller actions. For example, consider the following route template:

```
1. app.MapControllerRoute(  
2.     name: "expiration",  
3.     pattern: "expiration/{year}/{month}/{day}/{category}/{  
         slug}"  
4.     defaults: new { controller = "Products", action = "Expiration"  
         });
```

In this route template, **{year}**, **{month}**, **{day}**, **{category}**, and **{slug}** are route parameters. When a request matches this route, the values for these parameters will be extracted from the URL and passed to the corresponding action method in the **ProductsController**.

Attribute routing

While convention-based routing, as shown in the preceding section, is useful for many scenarios, ASP.NET Core also supports attribute routing. Attribute routing allows defining routes directly on your controllers and their actions, providing more granular control over routing behavior, eventually causing some confusion or mess around route strings if not well managed. Consider the following controller:

```
1. [Route("api/[controller]")]
```

```
2. public class ProductsController : ControllerBase
3. {
4.     [HttpGet("{id}")]
5.     public IActionResult Get(int id)
6.     {
7.         // Retrieve product by id
8.     }
9.
10.    [HttpPost]
11.    public IActionResult Create([FromBody] Product product)
12.    {
13.        // Create a new product
14.    }
15. }
```

In this example, the **ProductsController** is annotated with the **[Route("api/[controller]")]** attribute, specifying that all actions within this controller are prefixed with **/api/products**. Additionally, the **GET** action is annotated with **[HttpGet("{id}")]**, indicating that it handles HTTP **GET** requests with a route parameter named **id**.

Configuring and customizing routing

Configuring and customizing routing in ASP.NET Core allows developers to tailor the routing behavior of their applications to specific requirements. Whether it is defining route templates, applying route constraints, or implementing custom route handlers, ASP.NET Core provides a range of features to achieve flexible and efficient routing.

Defining route templates

Route templates serve as the blueprint for matching incoming URLs to controller actions. They consist of placeholders representing dynamic segments of the URL, along with optional route parameters. As we have seen in the previous paragraph, tokens enclosed in curly braces specify the path parameters that are assigned values when the path is matched. Multiple route parameters can be defined within a single route segment; however, each route parameter must be separated by a literal value.

The non-path parameter text (for example, {id}) and the path separator “/” must match the corresponding text in the URL exactly. The match is case-insensitive and based on the decoded form of the URL path. To match the delimiters of { or } literal path parameters, escape by doubling the character, such as {{ or }}.

Catch-all route template

When you use an asterisk (*) or double asterisk (**) as a prefix for a route parameter in ASP.NET Core, you get a peculiar behavior: they capture the residual portion of the URI that has not been matched by segments of previous routes. These special parameters, also known as **catch-all parameters**, allow you to dynamically manage various parts of the URI, making them particularly useful in scenarios where you need to manage dynamic or hierarchical URLs, such as in content management systems or URLs optimized for search engines (SEO).

Let us take the **blog/{**slug}** pattern for example. This pattern is designed to match any URI that begins with **blog/** followed by any value. When a match is found, the part of the URI after **blog/** is captured and assigned to the path parameter named **slug**.

Thanks to this, it is possible to effectively capture and process different parts of the URL path within the application logic, allowing for more flexible and dynamic management of requests.

Let us consider the following example:

```
1. app.UseRouting();
2.
3. app.MapControllerRoute(
4.     name: "products",
5.     pattern: "api/{controller}/{action = Index}/{id?}");
6. app.MapControllerRoute(
7.     name: "default",
8.     pattern: "{controller = Home}/{action = Index}/{id?}");
9. app.MapControllerRoute(
10.    name: "catchAll",
11.    pattern: "api/{**path}",
12.    defaults: new { controller = "Error", action = "Handle404" });
```

Catch-all parameters can also match an empty string. They handle

proper character escaping when the route is used to generate a URL, including path separators, as in the example route `foo/{*path}` with route values `{ path = "my/path" }` resulting in `foo/my%2Fpath`, note the escaped slash. For bidirectional path separators, use the route parameter prefix `**`, as in `foo/{**path}` with `{ path = "my/path" }` resulting in `foo/my/path`.

Route constraints and parameters

In RESTful APIs, route constraints ensure URL patterns match specific criteria, enhancing security and data integrity. Parameters, including path and query parameters, allow dynamic endpoints by passing variable data within the URL or request. This flexibility enables tailored client requests to specific resources, ensuring efficient and predictable API interactions.

Route file-path template

A simple controller able to manage the `path` route looks as follows:

```
1. public class TemplateController : Controller
2. {
3.     public IActionResult Render(string path)
4.     {
5.         // Logic to render the template based on the provided path
6.         return Content($"Rendering template for path: {path}");
7.     }
8. }
```

When defining a URL structure to include file names with optional extensions, it is important to consider how to handle this in the URL path. To make a parameter optional, we have to add a question mark (?) at the end of its name, for example, `id?`. For example, when you define a path as `template/{filename}.{ext?}`, you expect both the file name and extension to be present. However, if the extension is omitted in the URL request and only the file name is provided, the path will still match since the dot is considered optional, providing greater flexibility when defining URL paths. For instance, the following URLs match:

- `/documents/theFile.txt`

- `/documents/theFile`

The definition of the endpoint should look as follows:

```
1. app.MapControllerRoute(  
2.     name: "default",  
3.     pattern: "{controller = Home}/{action = Index}/{id?}");  
4.  
5. // Route with optional file extension  
6. app.MapControllerRoute(  
7.     name: "file",  
8.     pattern: "files/{filename}.{ext?}", // optional extension  
9.     defaults: new { controller = "Files", action = "Download" });
```

A theoretical controller that matches the above route should look as follows:

```
1. public class FilesController : Controller  
2. {  
3.     public IActionResult Download(string filename, string ext)  
4.     {  
5.         if (string.IsNullOrEmpty(ext))  
6.         {  
7.             // Logic for handling file download without extension  
8.             return Content($"Downloading file: {filename}");  
9.         }  
10.        else  
11.        {  
12.            // Logic for handling file download with extension  
13.            return Content($"Downloading file: {filename}.{ext}");  
14.        }  
15.    }  
16. }
```

Route parameters can be given default values by specifying them after the parameter name and separating them with an equal sign (=). For example, `{controller=Home}` establishes **Home** as the default value for the controller, which is used when the URL provides no value for that parameter. According to the two ways, optional values and default parameters, a default parameter always returns a value, and an optional parameter takes on a value only when present in the request URL.

Constraint resolver

Route parameters can also be subject to constraints that must be respected by the route value in the URL. By adding “:” and the constraint name after the parameter name, we can specify an inline constraint for the route parameter. Any arguments needed by the constraint are enclosed in parentheses (...) after its name. We can define multiple constraints inline by adding another “:” followed by the constraint name.

In ASP.NET Core, **IInlineConstraintResolver** is a service that resolves inline-defined route constraints within URLs. The **IInlineConstraintResolver** service receives the constraint name and arguments to create an **IRouteConstraint** instance, which is used to process the URL. For example, in the route **products/{weight:max(5)}**, the constraint specifies a maximum weight with argument 5.

Additionally, path parameters can be transformed using parameter transformers, which change the value of a parameter when generating links and matching actions and pages to URLs. Similar to constraints, you can add inline parameter transformers to a path parameter by placing “:” followed by the name of the transformer after the parameter name. For example, in the **products/{name:slugify}** route, you specify a transformer of type **slugify**.

Applying route constraints

Route constraints enable the definition of conditions necessary for a route to be deemed valid for an incoming request, ensuring that routes are matched solely based on meeting specified criteria, thereby enhancing the accuracy and reliability of routing. Take into account the following subsequent code:

```
1. app.MapControllerRoute(  
2.     name: "blog",  
3.     pattern: "blog/{year}/{month}/{day}/{slug}",  
4.     defaults: new { controller = "Blog", action = "Post" },  
5.     constraints: new  
6.     {  
7.         year = @"\d{4}",  
8.         month = @"\d{1,2}",
```

```
9.         day = @"\d{1,2}"
10.     });
```

In this example, route constraints are applied to the `{year}`, `{month}`, and `{day}` placeholders, ensuring that they match numeric values with specific patterns (e.g., year should be four digits, month and day should be one or two digits).

Attribute routing with templates

Using attribute routing offers a more declarative approach to defining routes directly on controller actions or on the controllers themselves using attributes. This simplifies routing configuration and improves readability, especially in cases where routes are closely tied to specific actions. It looks as follows:

```
1. [Route("api/[controller]")]
2. public class ProductsController : ControllerBase
3. {
4.     [HttpGet("{id:int}")]
5.     public IActionResult GetProduct(int id)
6.     {
7.         // Retrieve product by id
8.     }
9. }
```

In this example, the `[HttpGet("{id:int}")]` attribute specifies a route template with a constraint (`:int`) that restricts the `id` parameter to integer values only.

Proper routing configuration and customization in ASP.NET Core is a powerful tool for developing efficient and reliable web applications.

Parameter transformers

The parameter transformer in ASP.NET Core routing lets you customize how URL parameters are interpreted or transformed during routing. This can be particularly useful for implementing naming conventions or modifying routing behavior to suit specific needs. Parameter transformers can be applied to controller and action parameters, changing the value of a parameter before it is processed by the routing system.

A parameter transformer is a class that implements the **IOutboundParameterTransformer** interface. This interface requires implementing a **TransformOutbound** method, which is called by the routing system when a URL is generated, for example, via the link or redirect helper. The method accepts a value (the parameter to transform) and returns the transformed value as a string. Parameter transformers can be applied at various points in the application:

- **Attribute routes:** This can be applied directly in attribute routes using the **{parameter:transformer}** syntax in the route definitions.
- **Routing conventions:** Routing conventions can be applied globally or to specific actions or controllers. This is useful for consistently applying transformations across your application without having to change each route attribute individually.
- **Startup configuration:** When configuring routing in your application, you can specify parameter transformers for certain routes.

A common example of using parameter transformers is transforming strings from camelCase to kebab-case (or vice versa), making URLs more readable and SEO-friendly. For example, if you have an action parameter called **mySampleAction**, a transformer could change it to **my-sample-action** in the generated URL.

Following is a simple example of a transformer that converts the parameters from camelCase to kebab-case using RegEx:

```
1. public class KebabCaseParameterTransformer :  
    IOutboundParameterTransformer  
2. {  
3.     public string TransformOutbound(object value)  
4.     {  
5.         if (string.IsNullOrEmpty(value) ) return null;  
6.         return Regex.Replace(value.ToString()  
7.             , "([a-z])([A-Z])", "$1-$2").ToLower();  
8.     }  
9. }
```

After defining the transformer, we can apply it to routes by adding it to the routing conventions in your application in **Program.cs** or

registering it as part of a route attribute.

1. `builder.Services.AddRouting(options =>`
2. `options.ConstraintMap["kebab-case"]`
3. `= typeof(KebabCaseParameterTransformer));`

The ASP.NET Core employs parameter transformers to modify the URI that leads to a specific endpoint. For instance, these transformers alter the route values associated with an area, controller, action, and page to ensure a match.

Parameter transformers provide a powerful and flexible way to influence routing behavior in ASP.NET Core. They allow you to customize generated URLs based on specific needs or conventions.

Best practices and tips for routing

Effective routing management is critical to ensuring optimal performance and smooth navigation within an ASP.NET Core application. In this section, we will explore best practices and some helpful tips to maximize routing efficiency and scalability, thus improving the overall user experience.

Implementation of versioning

You might be tempted to use the `[ApiVersion("1")]` attribute for versioning, but the attribute is less useful than you might think. For the best use of versioning, we assume the following:

- Version routing is done through the route as opposed to through query strings like in public APIs (i.e., `/v1/blogs` vs. `/blogs?api-version=2022-10-21`) or with HTTP headers.
- Most APIs will stay in their **V1**, with each change usually being back and forward-compatible (i.e., a new thing is added, but the semantics stay the same).

This matches the experiences: usually, over the period of most APIs, there is rarely a need to introduce a **V2**, but when it is needed, it is of a big advantage to have **V1** and **V2** in the same project and gradually either phase out **V1** or support it for old external parties.

As such, it is recommended you place the **V1**/ version simply in the route on the controller and leave it there. You will either never

touch it again, or create a new class if you need a **V2/** controller. If you had used **ApiVersion**, you would still need to do that anyway.

Compare the **[ApiVersion("1")]** solution, as follows:

```
1. // bad example
2. [ApiVersion("1")]
3. [Route("v{version:apiVersion}/home")]
4. public class HomeV1Controller : Controller
5. {
6.     [HttpGet]
7.     public string Get() => "Version 1";
8. }
9. [ApiVersion("2")]
10. [Route("v{version:apiVersion}/home")]
11. public class HomeV2Controller : Controller
12. {
13.     [HttpGet]
14.     public string Get() => "Version 2";
15. }
```

With the preferred route-based solution, as follows:

```
1. // better solution
2. [Route("v1/home")]
3. public class HomeController : Controller
4. {
5.     [HttpGet]
6.     public string Get() => "Version 1";
7. }
8.
9. [Route("v2/home")]
10. public class HomeV2Controller : Controller
11. {
12.     [HttpGet]
13.     public string Get() => "Version 2";
14. }
```

If our APIs are implemented with the use of minimal API, we can use the same best practices we have seen in [Chapter 4, Designing RESTful APIs](#). In any case, we can define a static class containing the endpoints and versioning, as we have seen in the previous chapter,

and we can use it in both minimal API and controllers. The previous controller will be modified as follows:

```
1. [Route($"{Endpoints.Version}/{Endpoints.HomeApiEndpoint}")]
2. public class HomeController : Controller
3. {
4.     [HttpGet]
5.     public string Get() => "Version 1";
6. }
7.
8. [Route($"{Endpoints.Version2}/{Endpoints.HomeApiEndpoint}")]
9. public class HomeV2Controller : Controller
10. {
11.     [HttpGet]
12.     public string Get() => "Version 2";
13. }
```

Only to remember, the **Endpoints** static class has public const string members because the Route attribute should have constant values as attribute, which is shown as follows:

```
1. public static class Endpoints
2. {
3.     public const string Version = "V1";
4.     public const string VersionNext = "V2";
5.     public const string HomeApiEndpoint = "Product";
6. }
```

Keeping routes simple and predictable

When designing an application routing system, we should always keep it as simple and easily understandable as possible. Therefore, our routes should be intuitive and predictable for both the developers working on the project and the users navigating the application.

To accomplish this goal, it is important to adopt a design approach emphasizing clarity and consistency in defining application routes. This means creating routes that reflect the logical structure of the application and are easily understandable for both developers and users of the application.

An essential part of this approach is the use of clear and descriptive route patterns. This implies using meaningful names for the parameters and segments of the routes so that the purpose of each is immediately understandable. Additionally, it is important to avoid overly complex route configurations that could make the routing system difficult to understand and maintain over time.

Following RESTful conventions, if applicable, can also help keep routes simple and predictable. RESTful conventions offer a standardized approach to designing web APIs, which can make it much easier to understand and manage routes in the context of an ASP.NET Core application.

Finally, it is critical to document the routing logic within your application code. Providing comments or clear documentation can help developers understand the behavior and purpose of each route, making maintenance and debugging easier over time.

Let us see two examples of good code and bad code, as follows:

```
1. app.MapControllerRoute(  
2.     name: "ComplexRoute",  
3.     pattern: "{controller}/{action}/{id?}/{*catchall}",  
4.     defaults: new { controller = "Home", action = "Index" }  
5. );
```

This first example is not a good implementation of a route template. The route is complex to understand and maintain.

The following example, instead, is a good example of a maintainable and understandable code:

```
1. app.MapControllerRoute(  
2.     name: "SimpleRoute",  
3.     pattern: "{controller = Home}/{action = Index}/{id?}"  
4. );
```

Using attribute routing sparingly

It is good practice to limit the use of this approach when defining routes within an ASP.NET Core application. Instead of defining routes directly on top of controller methods via **[HttpGet]**, **[HttpPost]**, etc. attributes, it is recommended to use conventional routing, where routes are defined centrally in the **Program.cs** class

using the **MapRoute** or **MapControllerRoute** method in combination with **[Route]** attributes in controllers only for specific cases or particular requirements.

Overusing attribute routing can lead to less organized and difficult-to-maintain code, especially in large applications. Instead, keeping the route definition in one place makes it easier to maintain an overview of the application structure and consistently and efficiently make changes or extensions to the routing.

Take a look at the following code:

```
1. // Defining routing via attributes for all actions in the controller
2. [Route("api/[controller]")]
3. [ApiController]
4. public class ProductsController : ControllerBase
5. {
6.     // Defining the path of each action using routing attributes
7.     [HttpGet("getproducts")]
8.     public IEnumerable<Product> GetProducts()
9.     {
10.         // Logic to get products
11.     }
12.
13.     // Defining the path of each action using routing attributes
14.     [HttpPost("addproduct")]
15.     public IActionResult AddProduct(Product product)
16.     {
17.         // Logic to add a product
18.     }
19. }
```

In this code, each route is defined above each method creating confusion and possibility of mistakes. This code is considered **bad practices**.

In the following code, we have properly configured the route in our **Program.cs** by writing:

```
1. // Program.cs
2. app.UseRouting();
3. app.MapControllerRoute(
```


4. `name: "default",`
5. `pattern: "{controller = Products}/{action = Index}");`

Additionally, following is the corresponding controller:

```
1. // ProductsController.cs
2. [ApiController]
3. [Route("api/[controller]")]
4. public class ProductsController : ControllerBase
5. {
6.     // GET: api/Products
7.     [HttpGet]
8.     public IActionResult Index()
9.     {
10.         // Logic to get all products
11.         return Ok("All products");
12.     }
13.     // GET: api/Products/5
14.     [HttpGet]
15.     public IActionResult GetProduct(int id)
16.     {
17.         // Logic to get product by ID
18.         return Ok($"Product with ID {id}");
19.     }
20.     // POST: api/Products
21.     [HttpPost]
22.     public IActionResult AddProduct([FromBody] Product
product)
23.     {
24.         // Logic to add a product
25.         return Ok("Product added successfully");
26.     }
27. }
```

By considering convention-based routing for most scenarios and reserving attribute routing for exceptional cases where more granular control is needed, you will end up with more readable and lightweight code that is easy to maintain and evolve. This practice is not for minimal API approach because minimal APIs are designed for single actions and falls into the exceptional cases seen before.

Leveraging route constraints wisely

Applying route constraints in the routing of a web application serves to define precise criteria that the route parameters must satisfy, improving security and performance. However, excessive or inappropriate use of constraints can introduce complexity into the routing logic, making the code more difficult to maintain and understand. Furthermore, particularly restrictive or computationally intensive constraints can negatively impact performance, increasing request processing time rather than reducing it. This approach can also limit the application's flexibility in handling variations in input data, restricting the ability to adapt to unanticipated use cases. Finally, relying solely on constraints to validate inputs without implementing clear error handling can deteriorate the user experience, leaving users without understandable feedback in the event of invalid input data. A thoughtful use of path constraints is therefore essential to prevent these tools, although useful, from becoming a source of inefficiencies.

In short, wisely using path constraints is like being the best navigator for your application's requests, ensuring each one finds its way in the safest and most efficient manner possible.

Let us take a look at the following code:

```
1. // Bad code
2. public class YearConstraint : IRouteConstraint
3. {
4.     public bool Match(HttpContext httpContext
5.         , IRouter route
6.         , string routeKey
7.         , RouteValueDictionary values
8.         , RouteDirection routeDirection)
9.     {
10.         // Use of implicit types and lack of error handling
11.         var value = values[routeKey];
12.         if (int.TryParse(value?.ToString()
13.             , out var year))
14.         {
15.             // Hardcoded year range
16.             return year > 1900 && year < 3000;
```

```

17.     }
18.     return true; // Returns true by default
19. }
20. }
21. // Program.cs
22. app.UseRouting();
23. app.MapControllerRoute(
24.     name: "default",
25.     pattern: "{controller=Home}/{action=Index}/{year:int?}",
26.     defaults: new { year = DateTime.UtcNow.Year },
27.     constraints: new { year = new YearConstraint() });

```

In this example of bad code, a **YearConstraint** path constraint is defined, but the **Match** method has several problems. We can clearly see problems such as a lack of error handling, hardcoded values, the check of the years' range being too wide and inefficient, and confusing logic, which make the code less readable, less secure, and more difficult to maintain. The approach clearly demonstrates how neglect of fundamental programming principles can have a substantial negative impact on the final outcome of a software project.

On the other side, the following example will be better:

```

1. // Good code: Defining a route constraint for a route parameter
2. public class YearConstraint : IRouteConstraint
3. {
4.     public bool Match(HttpContext httpContext
5.         , IRouter route
6.         , string routeKey
7.         , RouteValueDictionary values
8.         , RouteDirection routeDirection)
9.     {
10.         int year;
11.         if (values.TryGetValue(routeKey, out object value)
12.             && int.TryParse(value?.ToString(), out year))
13.         {
14.             return year >= 2000 && year <=
DateTime.UtcNow.Year;
15.         }

```

```
16.     return false;
17. }
18. }
```

The routing in the **Program.cs** will be the same.

This example is considered good code. We define a **YearConstraint** path constraint that tests whether the year parameter is an integer between 2000 and the current year. This constraint is then applied to the default route in the application through the use of **MapControllerRoute** within **Program.cs**. The **year** parameter is also set to a default value of the current year if it is not specified in the URL.

Thoughtfully organizing controllers and actions

In ASP.NET Core application development, optimal organization of controllers and actions constitutes a fundamental practice that directly influences the maintainability, scalability, and clarity of application routing. Controllers, acting as intermediaries between the data model and the user-presented view, along with the actions they encapsulate, should be structured following software design principles such as Single Responsibility and Separation of Concerns (see [Chapter 1, Introduction to ASP.NET Core](#), for details about SOLID principles). This targeted organization allows for the functionalities to be isolated into well-defined logical blocks, thereby facilitating code reusability, maintenance, and the introduction of new features without negatively impacting the existing architecture. Moreover, such structuring promotes a more intuitive and manageable routing configuration, essential for efficiently directing user requests to the correct action handlers. Adopting a systematic and well-planned approach to organizing controllers and actions within ASP.NET Core thus results in optimized performance and greater ease of evolution and understanding of the web application.

Two actions to take care of this subject are as follows:

- Group related controllers and actions logically to facilitate easier navigation and maintenance.
- Consider using areas or namespaces to organize controllers into distinct functional areas or modules.

Implementing RESTful routing

Implementing RESTful routing in an ASP.NET Core application involves designing the application's endpoints to align with REST principles (see [Chapter 4, Designing RESTful APIs](#)). This approach aims to provide a standardized way to define interactions between clients and servers, making APIs intuitive and easy to use. RESTful routing is centered around the concept of resources, and actions are performed on these resources using standard HTTP methods such as GET, POST, PUT, DELETE, etc.

There are two recommendations around this best practice:

- Follow RESTful principles when designing API routes to create predictable and intuitive APIs.
- Use HTTP verbs (GET, POST, PUT, DELETE) to indicate the intended action for each route, rather than relying solely on route templates.

Handle missing routes gracefully

In web development, gracefully handling **Route Not Found** scenarios is essential for enhancing user experience and maintaining system security. This involves designing web applications to return informative and user-friendly error messages instead of generic **404 Not Found** responses when users request undefined routes. By implementing custom error responses that guide users toward potential solutions or more information, developers can improve navigability and user satisfaction. Additionally, logging these incidents provides valuable insights for identifying usability issues or potential security threats. A balanced approach is required to ensure these error messages are helpful without disclosing sensitive information that could be exploited. Adopting such practices not only elevates the user experience by providing clear guidance in the face of errors but also fortifies the application's security posture by mitigating information leakage risks.

A poor implementation of handling a not found route (404 error) in a web application might look as follows:

1. `public class HomeController : Controller`
2. `{`
3. `public IActionResult Index()`

```

4.     => View();
5.
6.     public IActionResult About()
7.     => View();
8.     // Attempt to handle a not found route directly in each action
9.     public IActionResult Contact(string path)
10.    {
11.        if (path != "expected-path")
12.        {
13.            // Logging the error - This should not happen in a
specific action
14.            Console.WriteLine($"Path not found: {path}");
15.            // Directly returns a 404 status code without a useful
page
16.            Response.StatusCode = 404;
17.            return Content("Page not found.");
18.        }
19.        return View();
20.    }
21. }

```

This approach exhibits several issues. Each action controller attempts to handle not found routes individually, leading to code duplication and difficult maintenance. Furthermore, this approach disperses the error handling logic throughout the application, reducing consistency.

Using **Console.WriteLine** for logging errors is not recommended for a professional web application. A more robust logging framework that supports log levels, filtering, and multiple log destinations (files, console, external services, etc.) would be preferable. Returning a 404 status code directly with a simple text message is not user-friendly. It does not provide any context, help, or links to navigate to other parts of the application. Not having a custom error page makes the user experience less friendly. A good error page should provide useful information on what to do next while maintaining the site's design and navigation.

To correct the issues and improve upon the previous example, you can centralize error handling and leverage ASP.NET Core's built-in features to gracefully handle not found routes and other errors.

Following is an example of a good implementation practice in **Program.cs**:

- Use the **UseStatusCodePagesWithReExecute** middleware to catch responses with status codes that do not include a body, such as 404, and re-execute them to a dedicated controller/action.
- Configure a global exception handler for non-development environments.

```
1. // Configure the HTTP request pipeline.
2. if (app.Environment.IsDevelopment())
3. {
4.     app.UseWebAssemblyDebugging();
5. }
6. else
7. {
8.     app.UseExceptionHandler("/Error",
        createScopeForErrors: true);
9.     // The default HSTS value is 30 days. You may want to
        change...
10.    app.UseHsts();
11.    // Handles status codes like 404
12.    app.UseStatusCodePagesWithReExecute("/Error/
        StatusCode", "{code = {0}}");
13. }
14. app.MapControllerRoute(
15.     name: "default",
16.     pattern: "{controller = Home}/{action = Index}/{id?}");
```

To properly manage the route, let us implement an error controller that will be able to centralize the management of errors. In that controller, we have to implement specific actions to handle different types of errors, including status errors like 404. The controller will look as follows:

```
1. public class ErrorController : Controller
2. {
3.     public IActionResult Index()
4.         => View();
5.     [Route("Error/StatusCode")]
```

```
6. public IActionResult StatusCodeHandler(int code)
7. {
8.     // Could be handle different status codes
9.     if (code == 404)
10.    {
11.        ViewBag.ErrorMessage = "Sorry, the requested page
was not found.";
12.    }
13.    return View("NotFound");
14. }
15. }
```

This approach centralizes error handling, making the code cleaner and more maintainable and improving user experience with customized and informative error pages. Error logging can be managed at the middleware or service level, using an appropriate logging framework to provide detailed error information without cluttering controllers with logging logic.

Conclusion

In this chapter, we understood the robust routing capabilities of ASP.NET Core, from basic configurations to more advanced configurations. By leveraging conventional and attribute routing, developers are able to guide the flow of application requests with precision, ensuring URLs remain clear and navigable.

Exploring RESTful route design, route constraints, and custom handlers highlights the importance of routing in modern web development. It is clear that a thorough understanding of routing is essential to building scalable and maintainable applications. As developers continue to innovate, the insights gained from ASP.NET Core routing will undoubtedly be a critical building block in developing dynamic, user-centric web applications. One of the most important concepts after routing are the middleware and extensibility.

In the next chapter we will clarify how middleware allows developers to insert custom logic at various points in the request pipeline.

Points to remember

- **Use attribute routing for clarity and control:** Attribute routing in ASP.NET Core provides precise control over your application's URL patterns, enhancing readability and maintainability of routes.
- **Leverage route constraints to validate requests:** Implementing route constraints ensures that only valid requests are processed, improving application security and reducing unnecessary server load.
- **Adopt RESTful principles for API design:** Designing routes with RESTful principles in mind promotes standardized and intuitive API endpoints, facilitating easier integration and consumption by clients.
- **Test your routes thoroughly:** Regular testing of your routing configuration is essential to catch potential issues early, ensuring that all parts of your application are accessible and function as intended.

Multiple choice questions

1. Which of the following is not a standard HTTP method used in RESTful APIs?
 - a. GET
 - b. POST
 - c. SEND
 - d. DELETE
2. In ASP.NET Core, what attribute is used to specify that a controller is intended to serve HTTP API responses?
 - a. [ApiController]
 - b. [WebController]
 - c. [HttpClient]
 - d. [ServiceController]
3. What does the IActionResult interface in ASP.NET Core represent?

- a. A database result set
 - b. An HTTP response
 - c. A serialized JSON object
 - d. A set of routing rules
4. Which feature of ASP.NET Core allows developers to define routes using attributes on controller actions?
- a. Conventional routing
 - b. Attribute routing
 - c. Dynamic routing
 - d. Static routing
5. When a route is not found, what is the best technique to adopt?
- a. Redirect to action
 - b. Raise an error
 - c. Configure a catchall route
 - d. Manage a JSON file exception

Answer key

Key terms

- **Endpoints:** This refers to a specific location within a web application identified through a specific and strong combination of URL and HTTP methods.
- **Route:** URL pattern that is mapped to a handler and is used to determine how an incoming web request is dispatched to an endpoint.
- **Route templates:** A map incoming URLs to actions using placeholders and optional parameters, with curly-braced tokens defining path parameters assigned upon URL matching.

CHAPTER 6

Middleware and Extensibility

Introduction

The middleware in ASP.NET Core acts as an intermediary between user requests and server responses. Review, manage or modify both incoming and outgoing requests within our web application. The order of the middleware in the request processing pipeline is of fundamental importance as it determines how requests and responses are handled.

This chapter explores the key concepts of architecture and in the design of modern software developed with ASP.NET Core. We will start with the concepts of **dependency injection (DI)**, a fundamental principle that improves the modularity and testability of applications. We will analyze the fundamental concepts of AI and examine how ASP.NET Core implements a version of it.

Next, we will analyse Middleware, listing both the built-in and available middleware in the framework and how to build custom middleware.

Structure

This chapter will cover the following topics:

- Dependency injection concepts
- Implementation of dependency injection in ASP.NET Core
- Understanding middleware components in ASP.NET Core
- Middleware ordering
- Creating custom middleware
- Extending the dependency injection

Objectives

In the chapter, we move from the fundamental concept of DI to the complex operation of the middleware pipeline in ASP.NET Core. The goal is to connect theory with practice, starting with the DI's role in promoting a modular architecture and then delving into how middleware functions as the backbone for processing HTTP requests. The main objective of the chapter is precisely the in-depth analysis of the creation of customized middleware, underlining the usefulness of customization and the importance of being able to short-circuit the HTTP request when necessary.

Dependency injection concepts

A dependency describes a relationship in which one entity relies on another for its functionality, existence, or operation, manifesting itself in various fields such as technology, biology, and social sciences. In biology, it might describe ecological relationships such as parasitism, in which one organism depends on another for survival. In the social sciences, dependencies can exist in relationships between individuals, groups, or countries, highlighting economic, social, or psychological dependence. In software, dependency means that a class object depends on another object. Following is the class:

```
1. public class MessageModel
2. {
3.     public void WriteMessage(string message)
4.     => Console.WriteLine($"This is the message: {message}");
```

5. }

This class can be instantiated, created, and can be called from another class as follows:

```
1. public class MessagesWrapper
2. {
3.     private readonly MessageModel messageModel = new
   MessageModel();
4.     public void WriteAll()
5.         => messageModel.WriteMessage("Parent message");
6. }
```

As we can see, the **MessageModel**, is a dependency of **MessagesWrapper**. In this case, we say that **MessagesWrapper** has a direct dependency on **MessageModel**. This kind of dependency has some problems, which are discussed as follows:

- **MessagesWrapper** is tightly coupled to **MessageModel**. This makes it challenging to replace **MessageModel** with a different implementation without modifying **MessagesWrapper**, limiting code flexibility.
- Since **MessagesWrapper** directly depends on **MessageModel**, reusing **MessagesWrapper** with different message handling mechanisms requires rewriting or extending the class, diminishing code reusability effectiveness.
- Any change in the logic or dependencies of **MessageModel** could necessitate corresponding changes in **MessagesWrapper** and other classes that depend on **MessageModel**, increasing the maintenance burden.
- Testing **MessagesWrapper** in isolation from **MessageModel** becomes more complex as it requires the instantiation of a real **MessageModel** or the use of advanced mocking techniques, complicating unit test writing.
- If **MessageModel** is extended with new functionalities or methods, **MessagesWrapper** may not be able to utilize them without direct modification, limiting the system's evolvability.

DI is a critical design pattern in modern application architecture, playing a pivotal role in developing applications with ASP.NET

Core. DI helps reduce the tight coupling between classes and their dependencies, making the code more manageable, extendable, and testable. In ASP.NET Core, DI is natively integrated, offering a robust platform for implementing easily testable and maintainable applications.

Refer to the following figure:

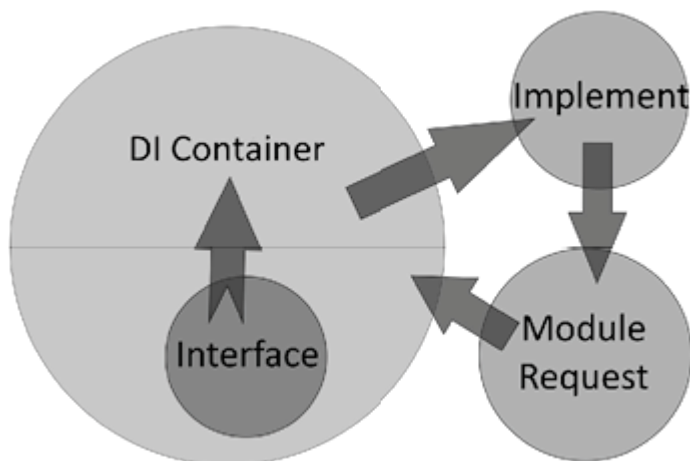


Figure 6.1: Dependency injection

Through DI, dependencies (like instances of classes for database access, logging, etc.) are no longer manually created inside the classes that use them. Instead, these dependencies are *injected* into the classes (or, for example, through constructors or via specific interfaces designed for each class), with the responsibility of their creation delegated to an external system known as the **DI framework**. All the instances are managed by the **DI container**. This approach not only promotes better isolation and separation of concerns but also simplifies application testing by allowing developers to replace real dependencies with mocks or stubs during testing.

Inversion of Control

The concept of **Inversion of Control (IoC)** is closely related to DI and forms its foundation. We have seen the IoC in [Chapter 1, Introduction to ASP.NET Core](#), speaking about SOLID principles. IoC reverses the traditional control flow of software, delegating the

management of dependencies to an external entity (the DI container). This means classes no longer control how and when to create their dependencies; instead, the framework takes on this responsibility, enhancing the modularity and flexibility of the application.

Service lifetimes

ASP.NET Core DI container manages three main types of service lifetimes, as follows:

- **Singleton:** The service instance is created only once for the entire application lifespan, promoting a shared state within the app.
- **Scoped:** A new instance of the service is created for each incoming request. It is useful for maintaining the state during a request, like user-specific data.
- **Transient:** New instances are created every time a service is requested, ideal for services that do not maintain state and are light to create and destroy.

Each duration option has a distinct purpose and has implications for performance, scalability, and application design. Singleton services are best used for scenarios with shared state across the entire application, scoped services are ideal for request-specific data maintenance and transient services cater to stateless operations that require frequent, lightweight creation and destruction. Selecting the appropriate service lifetime is a critical decision that impacts the overall architecture and efficiency of an ASP.NET Core application, ensuring that resources are optimally managed and services are delivered in a manner best suited to application needs.

Implementation of dependency injection in ASP.NET Core

DI involves automatically providing classes with the instances they need to perform their functions rather than hard-coding these instances within the classes. This process often occurs in a nested manner, where a needed instance itself requires other instances to work. A DI container manages this complexity by constructing and resolving a network of these dependencies, known as a dependency

graph or tree. This graph represents the relationships between various parts of an application, ensuring that each part gets the instances it requires. The primary benefits of DI include improved code modularity, easier testing through the use of mocks or stubs, and better maintainability by reducing tight coupling between components. We will see the testing strategies in [Chapter 12, Automated Testing](#).

Configuring dependency injection in ASP.NET Core

Configuration of DI in ASP.NET Core is a straightforward process that significantly enhances the modularity, testability, and maintainability of applications. ASP.NET Core comes with a built-in DI container that is configured in the **Program.cs** file. This built-in container supports constructor injection by default, allowing services to be registered and injected into the app without the need for external libraries. To set up DI, we start by defining the services and interfaces. These services could range from application-specific services like **data access layers (DAL)** or business logic layers to framework services such as logging, configuration, and environment contexts. Once defined, we proceed to register these services with the DI container in the **Program.cs** file. After registration, services can be inserted into the constructors of controllers, middleware, or other services by simply declaring them as parameters. The DI container instantiates and provides the requested service instances based on the registrations.

Let us see how our previous example changes shape using this technique by following these steps:

1. We have to create an interface for the model named **IMessageModel**, but in this implementation, the **WriteMessage** function will return the string message as follows:

```
1. public interface IMessageModel
2. {
3.     string WriteMessage(string message);
4. }
```

2. Our model implements this interface as follows:

```
1. public class MessageModel : IMessageModel
```



```

2. {
3.     public string WriteMessage(string message)
4.         => $"This is the message: {message}";
5. }

```

3. We declare this model in our DI in **Program.cs** file, as follows:

```

1.         builder.Services.AddSingleton < IMessageModel,
           MessageModel > ();

```

This line simply says that when a service defined by the **IMessageModel** interface is requested, DI returns a **MessageModel** instance so that each time we have a dependency that needs an **IMessageModel**, the DI container will return the **MessageModel**.

4. Let us rewrite the **MessagesWrapper** so that it accepts the parameter from DI.

```

1.     public class MessagesWrapper(IMessageModel
           messageModel)
2. {
3.     public string WriteAll()
4.         => messageModel.WriteMessage("Parent message");
5. }

```

5. After that, we will put this class into DI but in this case without an interface, as follows:

```

1. builder.Services.AddScoped < MessagesWrapper > ();

```

6. Finally, the minimal API that uses all the models is as follows:

```

1. app.MapGet("/", (MessagesWrapper messagesWrapper)
   =>
2. {
3.     return messagesWrapper.WriteAll();
4. })
5. .WithName("Get")
6. .WithOpenApi();

```

7. The output of this API will be as follows (using swagger):



Figure 6.2: Result of example

As we can see, the response body is what we expected.

Understanding middleware components

Middleware components are classes and logical structures in C# linked via a chain of responsibility design pattern. They manage and conclude the request pipeline according to their sequence. The order in which they are declared and executed is extremely important. By default, the middleware is set in the startup configuration (**Program.cs**), and the execution sequence is crucial. Requests are handled top-down, moving from the outside to the inside and then back again. Middleware is integrated into an application's pipeline and designed to manage requests and responses. Each component in the chain can decide to pass the request to the next component, enabling pre- and post-processing tasks. This structure allows for a modular approach to handling functionalities like authentication, logging, and encryption, enhancing an application's flexibility, scalability, and security.

The construction of this operational flow uses request delegates specifically assigned to supervise each HTTP interaction. These delegates act as critical components within the middleware pipeline, effectively controlling and directing the flow of HTTP requests and responses. Each delegate is responsible for processing the incoming request and passing it on to the next delegate in the sequence or terminating the process by generating a response. This mechanism ensures that each HTTP interaction is handled systematically, allowing for modular and efficient management of

web requests.

Let us see how the request pipeline works graphically as shown in [Figure 6.3](#):

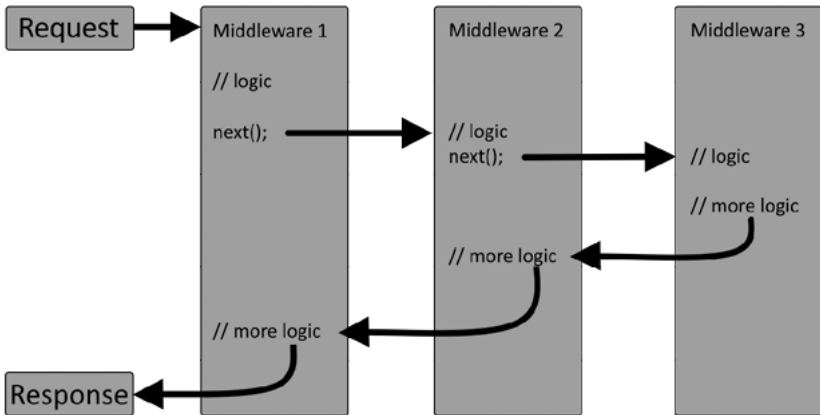


Figure 6.3: Request pipeline

These request delegates are set up through the use of methods such as **Run**, **Map**, and **Use**. A request delegate can be defined either as an inline anonymous method (termed inline middleware) or within a class designed for reusability. These anonymous methods or classes, known as middleware or middleware components, are responsible within the request chain for invoking the next component or interrupting the sequence. When a middleware interrupts the chain, it is termed terminal middleware because it prevents further middleware from processing the request.

A simple request delegate that handles a request is as follows:

```
1. app.Run(async context =>
2. {
3.     await context.Response.WriteAsync("Welcome to Chapter6!");
4. });
```

By using the **Use** method to wire multiple request delegates together, we will add more layers of middleware. Within each delegate, we will use the **next** parameter, which, in this context, acts as a reference to the next delegate in the pipeline. You can stop the pipeline from running by not invoking the **next** parameter. Typically, you can perform operations both before and after invoking the **next** delegate, as illustrated in the following example:

```

1. app.Use(async (context, next) =>
2. {
3.     // Here we can write on the Response.
4.     await next.Invoke();
5.     // Here we have not to write to the Response.
6. });
7.
8. app.Run(async context =>
9. {
10.     await context.Response.WriteAsync("Welcome to Chapter6!");
11. });

```

To see the result, run the code and call the application from a browser or an application. The first response will be our message.

Early termination of the request pipeline

When one delegate does not forward a request to the next, it is called an early termination (short circuit) of the request pipeline. This operation is often desirable as it allows you to avoid unnecessary work. For example, static file middleware can behave like terminal middleware by processing a request for a static file and terminating the rest of the pipeline early. Middleware is added to the pipeline before middleware that further interrupts processing continues to execute code after their **next.Invoke** statements. However, it is important to heed the following warning about attempting to write over a response that has already been posted.

Be careful not to invoke **next.Invoke** during or after sending the response to the client. Once an **HttpResponse** has been initiated, any modification may result in an exception. For example, setting the header and a status code throws an exception once the response has begun. Write in the body of the response after calling **next**, as follows:

- It might cause a protocol violation, such as writing beyond the declared **Content-Length**.
- It might corrupt the body format, like adding an **HTML** footer to a **CSS** file.
- **HasStarted** is a useful indicator to understand if the headers were sent or if they were written in the response body.

When routing identifies a matching endpoint, it typically allows the other middleware in the pipeline to execute before triggering the endpoint logic.

Tip: It is crucial not to invoke `next` after a response has begun sending to avoid exceptions and potential protocol violations.

Another way to short-circuit the pipeline is to call an API to make that. Services can optimize resource use by pre-filing requests that are already known at the beginning of the pipeline. To have routing directly trigger the endpoint logic and then conclude the request, you can use the **ShortCircuit** extension method. For example, there might be a specific route that does not require going through CORS or authentication middleware.

Following example immediately stops requests matching the **/short-circuit** route:

```
1. app.MapGet("/short-circuit", () => "Short  
   circuiting!").ShortCircuit();
```

There is another method to short circuit the requests: **MapShortCircuit**. This method is used as follows:

```
1. app.MapShortCircuit(404, "short-circuit-route", "other-route");
```

The **MapShortCircuit** method accepts an array of string able to short circuit all the param array routes in a single line. It can also short circuit files as **favicon.ico** or files for bots. If we place the **ShortCircuit** and **MapShortCircuit** methods before **UseRouting**, they will not affect the middleware.

Be careful applying these methods to endpoints carrying metadata such as **[Authorize]** or **[RequireCors]** will result in requests being blocked and an **InvalidOperationException** being thrown. This metadata is typically set through the use of the **[Authorize]** or **[EnableCors]** attributes or, alternatively, by calling the **RequireCors** or **RequireAuthorization** methods.

RUN delegates

When in the application pipeline we use a **Run** delegate, take note that the **Run** delegates do not receive a **next** parameter. The initial **Run** delegate within the pipeline is designed to be the concluding step, effectively ending the pipeline's process. This use of **Run**

follows an established convention. Additionally, some middleware components might offer **Run[Middleware]** methods, which are intended to operate at the end of the application pipeline.

```
1. app.Run(async context =>
2. {
3.     await context.Response.WriteAsync("Welcome to Chapter6!");
4. });
```

In this example, the **Run** delegate outputs "Welcome to Chapter6!" to the response before ending the pipeline's execution. Any subsequent **Use** or **Run** delegate will not be executed.

Building a middleware pipeline with web application

The ASP.NET Core request pipeline is composed of a series of request delegates executed sequentially. The process is depicted in a diagram where the execution thread follows the arrows through several middleware components, illustrating both the progression of a request through these components and its eventual response back to the client.

There are some key aspects to take into consideration, as follows:

- Each delegate can perform tasks before and after invoking the next delegate.
- Exception-handling delegates are placed early in the pipeline to catch exceptions from subsequent stages.
- The last middleware is called a **terminator** and reverses the path of the request, transforming it into a response.

Middleware ordering

The order of middleware components is crucial for security, performance, and functionality. ASP.NET Core apps demonstrate a comprehensive request processing pipeline, showcasing both standard and custom middleware's placement.

Refer to the following figure:

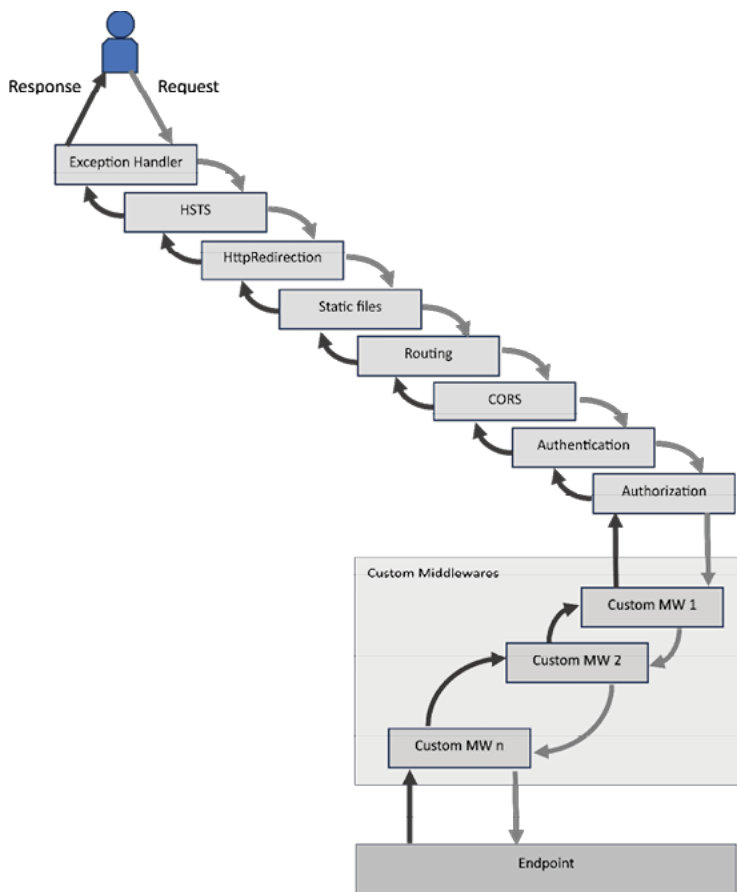


Figure 6.4: Requests/Responses middlewares pipeline

In the above diagram, we see the middleware pipeline and how, in a typical application, middlewares are ordered. By explicitly calling **app.UseRouting**, the routing middleware is executed at the point where it is placed in the middleware pipeline, not as the first middleware by default. Note the custom middlewares positioned just before the endpoint because the application middlewares manage critical subjects as authentication or authorization.

Middleware components in Program.cs

The sequence in which middleware components are added defines their execution order, impacting both request handling and response formation. Security-related middleware, for example, follows a recommended order for optimal protection. A pipeline

could be as follows:

```
1. var app = builder.Build();
2. if (app.Environment.IsDevelopment())
3. {
4.     app.UseMigrationsEndPoint();
5. }
6. else
7. {
8.     app.UseExceptionHandler("/Error");
9.     app.UseHsts();
10. }
11.
12. app.UseHttpsRedirection();
13. app.UseStaticFiles();
14. app.UseCookiePolicy();
15.
16. app.UseRouting();
17. app.UseCors();
18.
19. app.UseAuthentication();
20. app.UseAuthorization();
21. app.UseSession();
22. app.UseResponseCompression();
23. app.UseResponseCaching();
24.
25. app.Run();
```

In other applications, middlewares could be different or (in some particular cases) in a different order.

When creating a new web application with individual user accounts, it is important to understand the role and placement of middleware in the ASP.NET Core pipeline. Some middleware components, which might not be immediately necessary for a new project, are often commented out by default. This approach allows developers to tailor the application's middleware stack to their specific needs without cluttering the initial setup.

The order in which middleware components are added to the pipeline is crucial, as it can affect the application's behavior and

performance. While not every middleware needs to be in a specific sequence, certain ones do require a particular order. For example, **UseCors**, **UseAuthentication**, and **UseAuthorization** must be arranged precisely as listed to function correctly. Similarly, **UseCors** must precede **UseResponseCaching**, due to the dependencies between how CORS policies and response caching interact.

Another noteworthy point is the placement of **UseRequestLocalization**. This middleware should be positioned before any other that may examine the request's culture, such as **app.UseStaticFiles()**, ensuring the correct cultural settings are applied from the beginning of the request processing.

Lastly, the **UseRateLimiter** middleware demonstrates the importance of context when determining order. Specifically, when applying rate limiting to endpoint-specific APIs—indicated by the **[EnableRateLimiting]** attribute—**UseRateLimiter** must follow **UseRouting**. However, for global limiters, its position can be more flexible, even preceding **UseRouting**. This distinction highlights how middleware configuration can adapt based on the application's requirements, emphasizing the necessity of understanding each component's role and dependencies within the pipeline.

Common middleware scenarios

Middleware in ASP.NET Core apps cover a broad range of functionalities, from exception handling in different environments to session management and static file delivery. Each middleware plays a distinct role in the request-response lifecycle, contributing to the app's overall robustness and user experience. ASP.NET Core's middleware infrastructure provides a flexible and powerful mechanism to handle HTTP requests efficiently. In different environment types, we could have different configurations of middleware list stack. In the **Program.cs** file we can configure middleware for various typical application scenarios, detailing the handling of exceptions and errors differently based on the environment (Development, Staging or, Production).

Refer to the following code:

```
1. if (app.Environment.IsDevelopment())  
2. {
```

```
3. app.UseSwagger();
4. app.UseSwaggerUI();
5. app.UseDeveloperExceptionPage();
6. }
7. else if (app.Environment.IsStaging())
8. {
9.     app.UseSwagger();
10.    app.UseSwaggerUI();
11.    app.UseExceptionHandler("/Error");
12.    app.UseHsts();
13. }
14. else if (app.Environment.IsProduction())
15. {
16.     app.UseExceptionHandler("/Error");
17.     app.UseHsts();
18. }
```

Development environment

Here, it is beneficial to have detailed error reports for debugging purposes. Middleware like the Developer Exception Page can be configured to show detailed stack traces, error messages, and diagnostic information when an exception occurs. This detailed feedback helps developers quickly identify and fix issues. Furthermore, we can configure debugging tools like Swagger (automatically configured in case of ASP.NET Core Web API project).

Staging environment

This environment mirrors the production environment as closely as possible but is not accessible to the end users. Error handling in staging might still provide more information than in production but less than in development, aiming to test error-handling processes and logging without exposing sensitive debug information. Here, we can also configure debugging tools like Swagger.

Production environment

Finally, exposing detailed error information can be a security risk. Therefore, middleware configurations typically aim to log errors for

internal review while showing generic error messages to users. Middleware like exception handling middleware can be used to catch exceptions, log them for developers, and then redirect users to a user-friendly error page.

By carefully analyzing code, structuring the middleware pipeline and strategically ordering components, we can build secure and high-performance web applications.

Creating custom middleware

Each piece of middleware has the opportunity to process the incoming request and either pass it along to the next middleware in the sequence or terminate the process by generating a response. This setup offers a highly customizable request-handling mechanism, allowing software architects to strategically insert their own processing logic at various points in the request lifecycle.

When considering developing custom middleware, the decision typically arises from the need to implement application-specific logic or behavior that is not addressed by the existing middleware components provided by ASP.NET Core. Creating custom middleware is a strategic approach to encapsulate application-specific processing and cross-cutting concerns or to simplify logic repeated across multiple points of the application.

To implement custom middleware, one would define a class that includes either an **Invoke** or **InvokeAsync** method, with the latter being more common due to its support for asynchronous operations. This method must accept an **HttpContext** parameter, allowing it to interact with the request and response objects. The custom middleware is then registered in the application's startup process, ensuring it is executed as part of the HTTP request pipeline.

The following section contains a step-by-step guide to creating a simple logging middleware that logs the start and end of a request.

Define the middleware class

First, we need to create a class containing the custom middleware's logic. We will call this class **RequestLoggingMiddleware** and will be implemented, for instance, as follows:

```

1. public class RequestLoggingMiddleware(RequestDelegate next,
2. ILogger<RequestLoggingMiddleware> logger)
3. {
4.     public async Task InvokeAsync(HttpContext context)
5.     {
6.         // Log the start of the request
7.         logger.LogInformation("-----");
8.         logger.LogInformation($"Request: {context.Request.Path}");
9.         // Call the next middleware in the pipeline
10.        await next(context);
11.        // Log the completion of the request
12.        logger.LogInformation("Finished handling request.");
13.        logger.LogInformation("-----");
14.    }
15. }

```

Let us break it down for better understanding:

- **The primary constructor (Lines 1-2):** The **RequestLoggingMiddleware** class is defined with a constructor that takes two parameters: **RequestDelegate next** and **ILogger<RequestLoggingMiddleware> logger**. The **RequestDelegate** represents the next middleware in the request pipeline, allowing this middleware to pass the context to the next component once its work is done. The **ILogger** is a logging interface crafted specifically for this middleware class, which is used to log messages.
- **InvokeAsync method (Lines 3-14):** This asynchronous method, named **InvokeAsync**, is the method which takes an **HttpContext** object as its parameter. This method is crucial because it is where the middleware performs its operations on the HTTP context as it passes through the pipeline.
- **Starting request logging (Lines 6-8):** Before processing the request, the middleware logs the start. It prints a separator line for clarity and logs the request path. This helps in identifying which request is being processed and marks the beginning of a request log.
- **Calling the next middleware (Line 10):** The **await next(context);** the line is where the middleware passes

control to the next one in the pipeline. This line is essential for the request to continue its journey through the ASP.NET Core middleware pipeline. The operation is asynchronous, ensuring that the current middleware waits for the next middleware's processing to complete before proceeding.

- **Completion logging (Lines 11-13):** After the next middleware has processed the request (and potentially all subsequent middleware if this is not the last one), control returns to this middleware, and it logs the completion of the request handling. It marks the end of request processing with a message and another separator line for clarity.

Create an extension method for the middleware

To easily add your custom middleware to the application's request pipeline, you should create an extension method for the **IApplicationBuilder** interface.

Refer to the following code:

```
1. public static class RequestLoggingMiddlewareExtensions
2. {
3.     public static IApplicationBuilder UseRequestLogging(this
4.         IApplicationBuilder builder)
5.         =>
6.     builder.UseMiddleware<RequestLoggingMiddleware>();
7. }
```

This extension method is named **UseRequestLogging** for **IApplicationBuilder** within a static class **RequestLoggingMiddlewareExtensions**. This method enables easy addition of **RequestLoggingMiddleware** to the ASP.NET Core middleware pipeline with a concise and readable syntax. By invoking **builder.UseMiddleware<RequestLoggingMiddleware>()**, it registers our custom middleware **RequestLoggingMiddleware**, facilitating a streamlined middleware setup.

Register the middleware in the pipeline

Finally, incorporate the new middleware into the application's request processing pipeline by adding it to the program file.

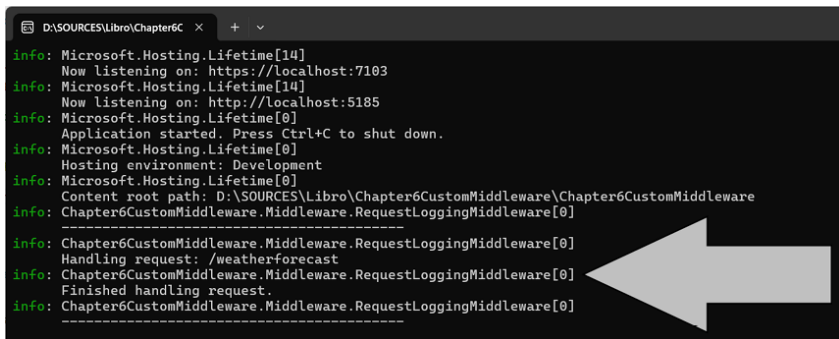
Refer to the following code:

1. *// Use the custom middleware*
2. `app.UseRequestLogging();`

If the extension method is too much (as in this example) for the involved service, we can register the middleware directly in **Program.cs** as follows:

1. *// Register the custom middleware in Program.cs*
2. `app.UseMiddleware<RequestLoggingMiddleware>();`

This custom middleware, when added to the ASP.NET Core application, will log the start and finish of handling each request, providing visibility into the request processing pipeline. This example can be expanded or modified to fit more complex scenarios, such as performing specific actions based on request headers or path. So, the output will be as follows:



```
D:\SOURCES\Libro\Chapter6C x + v
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:7103
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5185
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: D:\SOURCES\Libro\Chapter6CustomMiddleware\Chapter6CustomMiddleware
info: Chapter6CustomMiddleware.Middleware.RequestLoggingMiddleware[0]
      Handling request: /weatherforecast
info: Chapter6CustomMiddleware.Middleware.RequestLoggingMiddleware[0]
      Finished handling request.
info: Chapter6CustomMiddleware.Middleware.RequestLoggingMiddleware[0]
      -----
```

Figure 6.5: Custom middleware example output

As we can see in this case, we log directly on console, but is possible log it into a simple text file or a database.

Short circuit the pipeline

As we have seen in paragraph *Early termination of the request pipeline*, in case we want to short circuit the pipeline in case of a well-determined event, we can wrap it and give an immediate response to the caller. In our following case, we can figure out a simple case in which the incoming request starts with a precise address path.

We can write a new middleware that short circuits the pipeline and gives an immediate response named

FailedRequestWrapperMiddleware, as follows:

```
1. public class FailedRequestWrapperMiddleware(RequestDelegate
   next
2.     , ILogger<ImmediateResponseMiddleware> logger)
3. {
4.     public async Task InvokeAsync(HttpContext context)
5.     {
6.         if (!context.Request.Path.StartsWithSegments("/api/v1/"))
7.         {
8.             context.Response.StatusCode = 404; // Example status
   code
9.             await context.Response.WriteAsync("Custom Not Found
   Response");
10.            logger.LogWarning($"404 Not Found: {context.Request.
   Path}");
11.        }
12.        else
13.        {
14.            logger.LogInformation($"Request as: {context.Request.
   Path}");
15.            await next(context);
16.        }
17.    }
18. }
```

Initially, it checks if the incoming request path starts with a specific root address by using **context.Request.Path.StartsWithSegments("/api/v1/")**. If the request path does not start with **/api/v1/**, the middleware:

- Sets the response status code to **404** indicating that the requested resource was not found.
- Sends a custom response back to the client using **await context.Response.WriteAsync("Custom Not Found Response")**. This text is what the client will see in the response body.

Otherwise, if the condition is true, that is to say, if the request path indeed starts with **/api/v1/**, the middleware proceeds with two actions, as follows:

- Logs an informational message indicating the path of the received request.
- Calls **await next(context);** to pass control to the next middleware in the pipeline, allowing the request to continue its processing journey.

This custom middleware acts as a filter for incoming requests. If a request is not directed to an endpoint under `/api/v1/`, it treats it as invalid, immediately returns a 404 response with a custom message and logs a warning. For valid requests (i.e., those starting with `/api/v1/`), it logs an informational message and allows them to proceed through the pipeline to be further processed by subsequent middleware. This can be particularly useful for ensuring that only requests to desired API paths are processed while all others are promptly rejected.

Extending the dependency injection

Expanding the capabilities of an ASP.NET Core application often involves more than just configuring services and implementing middleware; it also entails enhancing the application's DI framework to suit complex needs. ASP.NET Core's built-in DI container is designed for simplicity and ease of use, but certain scenarios may require advanced configuration or the extension of the DI capabilities to ensure optimal application design and functionality.

Understanding the need for extension

The concept of extending the built-in DI container in ASP.NET Core stems from the need to address more complex and specific application requirements that go beyond the scope of what the framework's default DI container can handle. The built-in DI container is designed to be lightweight and straightforward, primarily supporting the essential DI features such as registering services with various lifetimes (singleton, scoped, transient) and resolving these services where needed. This simplicity is sufficient for many applications, but as projects grow in complexity, developers often find themselves needing more advanced features.

Deciding to manually extend the capabilities of a DI container can

be motivated by several factors, each critical to the seamless functioning and optimization of applications. For starters, the challenge of managing advanced object lifecycles stands out. The built-in service lifetimes provided might fall short in offering the nuanced control developers seek over how objects are created and disposed of, necessitating a more tailored approach to lifecycle management. Furthermore, integrating third-party libraries into an application is not always straightforward. These libraries often bring their own dependency injection configurations, requiring a bespoke setup that the default DI container might not readily accommodate, such as particular registration sequences or specialized Factory methods for service resolution.

Another reason is the need to meet complex configuration requirements. Applications may need to configure services based on runtime data or complex logic, a task that is beyond the capabilities of the standard DI container. Likewise, applications designed to serve multiple tenants, such as those on software-as-a-service platforms, face unique challenges. They must define the scope of services not only based on individual requests but also on tenants, requiring dynamic selection of the service implementation based on the requesting tenant, a scenario not directly supported by the conventional DI container.

Finally, the search for performance optimization cannot be overlooked. In scenarios where application demands are high, the performance of the integrated DI container may not be sufficient, prompting developers to look for alternatives that promise a significant increase in efficiency. Overall, these reasons highlight the need for manual extensions to DI container functionality, tailoring them to meet the specific requirements and challenges of modern application development.

Following is a simple example of a custom logger class. In the following example, we can find a class able to log (in some way) a message we will pass as an argument:

```
1. public class CustomLogger
2. {
3.
4.     public void Log(string message)
```

```
5.         => Console.WriteLine($"Custom Logger: {message}");
6.     }
7. }
```

After we have properly defined our class, we have to build it through the DI of ASP.NET Core. We can simply use the DI as follows:

```
1. builder.Services.AddSingleton<CustomLogger>();
```

In a different way, if our class needs special parameters to be built that cannot be retrieved from DI. Let us see it as follows:

```
1. public class CustomLogger
2. {
3.     public CustomLogger(LoggerConfig config)
4.     {
5.         // Imagine some complex initialization here using
LoggerConfig
6.     }
7.
8.     public void Log(string message)
9.     {
10.        Console.WriteLine($"CustomLogger: {message}");
11.    }
12.
13. }
14.
15. public class LoggerConfig
16. {
17.
18.     public string ConfigParameter { get; set; }
19.     // Other complex configuration properties
20.
21. }
```

In this case, we need a **LoggerConfig** that (in our imagination) cannot be built from DI.

The only way to build properly our logger is to write an extension method able to properly build the class, as follows:

```
1. public static class ServiceCollectionExtensions
```

```

2. {
3.     public static IServiceCollection AddCustomLogger
4.         (this IServiceCollection services)
5.     {
6.         // Retrieve LoggerConfig from IConfiguration
7.         var config = new LoggerConfig();
8.         var configuration = services.BuildServiceProvider()
9.             .GetRequiredService<IConfiguration>();
10.        configuration.GetSection("LoggerConfig").Bind(config);
11.        // Register CustomLogger with the configuration
12.        services.AddSingleton<CustomLogger>
13.            (provider => new CustomLogger(config));
14.        return services;
15.    }
16. }

```

As we can see, the **LoggerConfig** is retrieved from a JSON configuration. Now we have only to add this extension method to our **Program.cs** as usual.

In this way, we will extend our DI with some custom extension methods able to make the dirty job easier.

Keep in mind that the use of an extension method is not essential. It is possible to include the contents of the extension method directly within the **Program.cs** file. However, as mentioned in earlier chapters, doing so would contribute to cluttering our **Program.cs**, which is something we aim to avoid.

Strategies for Extending DI

Extending DI involves using various techniques to enhance the functionality and flexibility of the DI container. Key strategies include selecting an appropriate custom Factory or applying the options pattern.

Custom service factories

For complex object creation logic that goes beyond constructor injection, we can register custom Factory methods. These methods can encapsulate the logic needed to instantiate services with complex dependencies or configurations not suited to constructor

injection.

Consider the following code:

```
1. services.AddTransient<MyService>(provider =>
2. {
3.     var dependency = new MyDependency(complexParameters);
4.     return new MyService(dependency, "parameterValue");
5. });
```

In this case, the code shows how to handle complex dependencies (**MyDependency** in this case) by manually instantiating them and passing them to the service's constructor along with other necessary parameters. This pattern is useful for controlling the instantiation process of services with complex dependencies or when specific construction logic is required.

Multiple instances of the same interface

In a similar way to the previous example, we can have already registered multiple instances of the same interface because each implementation could have little differences (maybe inherits from another class or implements a more specific interface and so on). In this scenario, we have to distinguish each class and retrieve from DI the one we need. If we register the following instances:

```
1. builder.Services.AddScoped<IMyInterface,
    ImplementationA>();
2. builder.Services.AddScoped<IMyInterface, ImplementationB>();
3. builder.Services.AddScoped<IMyInterface,
    ImplementationC>();
4. var app = builder.Build();
5. var implementation =
    app.Services.GetService<IMyInterface>();
```

The variable **implementation** that we will have, will always be the instance of **ImplementationC** class because DI as default will return the last registered instance. If we would have the **ImplementationA**, we have to follow these steps:

```
1. var implementations =
    app.Services.GetServices<IMyInterface>();
2. var implementationA =
    implementations.OfType<ImplementationA>()
```

```
3. .FirstOrDefault();
```

Take note of the methods **GetService** and **GetServices**. The first one retrieves an instance of our interface, and the other method (plural) retrieves the whole list of registered services with the required interface.

The options pattern

ASP.NET Core's configuration system can be closely integrated with DI to dynamically register services based on configuration values. This allows for flexible application behavior that can be modified without changing the codebase. We can defer to the class creation the duty of his behavior as follows:

```
1. builder.Services.Configure<MyServiceConfig>  
2. (options =>  
   builder.Configuration.GetSection("MyServiceConfig")  
3. .Bind(options));
```

Then we can inject it in our class as follows:

```
1. public class MyService  
2. {  
3.     private readonly MyServiceConfig settings;  
4.     public MyService(IOptions<MyServiceConfig> settings)  
5.         => this.settings = settings.Value;  
6. }  
7.
```

Based on this implementation, we can control the behavior of multiple classes through a single, straightforward approach. This pattern is called **Option Pattern**.

Best practices for extending dependency injection

The built-in dependency injection, is easy to use and very versatile to bend to our needs. But we have to be careful using this technique. While extending DI can provide powerful capabilities, it is crucial to keep the DI configuration as simple and transparent as possible. Overly complex DI setups can make the application harder to understand and maintain.

Following are two main topics to ensure a clear and maintainable implementation:

- **Understand the trade-offs:** Introducing third-party DI containers can add advanced features but may also introduce additional dependencies and complexity. Evaluate the benefits against the potential drawbacks to ensure that the choice aligns with the application's requirements and long-term maintenance considerations.
- **Leverage built-in features first:** Before extending the DI container or integrating a third-party container, explore whether the built-in features of ASP.NET Core's DI can meet your needs. Often, creative use of Factory methods, lifecycle management, and configuration integration can provide the required functionality without additional complexity.

Extending the dependency injection capabilities in ASP.NET Core applications allow developers to tailor the DI framework to their specific needs, enabling more sophisticated service registration and resolution strategies. By carefully considering the application's requirements and maintaining a balance between advanced features and simplicity, developers can create flexible, maintainable, and scalable applications that leverage the full power of ASP.NET Core's DI system.

Conclusion

Middleware is the cornerstone of request processing in ASP.NET Core, offering a powerful and flexible way to build and configure web applications. By understanding the basics of middleware, how to configure it, and the capabilities of both built-in and custom middleware, developers can effectively manage how their applications handle HTTP requests and responses. This foundation sets the stage for further exploration into more advanced middleware concepts and configurations, which continue to underscore the extensibility and modularity of the ASP.NET Core framework.

Building on this knowledge, the next chapter will explore fundamental concepts guiding robust and scalable architecture design, ensuring maintainability and performance in software development.

Points to remember

- **Middleware ordering:** The order in which middleware components are added determines their execution sequence, affecting how requests are handled and responses are formed. The order is extremely important.
- **DI:** DI is a software design pattern that involves automatically supplying objects of a class with the instances of other objects they depend on, instead of having them construct those objects internally.
- **Services lifetime:** Service life affects the performance and design of ASP.NET Core. They are **Singletons** that handle shared state, **Scoped** that services handle data per request and **Transients** that support light and frequent operations.
- **Request pipeline:** The ASP.NET Core request pipeline is a sequence of request delegates depicted in a diagram illustrating the flow through middleware components and the final response to the client.
- **Custom middleware:** We use custom middleware typically when we want to execute custom special code before and after the next component in the pipeline.
- **Short-circuit the pipeline:** A delegate stopping a request is called short-circuiting, used for responses like page not found or static files.

Multiple choice questions

1. What is middleware in ASP.NET Core?
 - a. Software design pattern for database management
 - b. A software component that is executed in the request pipeline of an ASP.NET Core application.
 - c. A client-side JavaScript framework.
 - d. A type of service offered by the cloud.
2. Which of the following is not a common middleware in ASP.NET Core?
 - a. Authentication middleware

- b. CORS middleware
 - c. FTP request middleware
 - d. Static files middleware
3. In custom middleware, what method signature is commonly used?
- a. public void Configure(IApplicationBuilder app)
 - b. public void ConfigureServices(IServiceCollection services)
 - c. public async Task InvokeAsync(HttpContext context)
 - d. public IActionResult Index()
4. Which service lifetime does not exist in ASP.NET Core's built-in DI container?
- a. Singleton
 - b. Scoped
 - c. Transient
 - d. PerThread

Answer key

Key terms

- **Middleware:** Components that handle requests and responses within the application pipeline, enabling functionalities like authentication and routing. The sequence is very important.
- **HTTP request pipeline:** The sequence through which an HTTP request flows in ASP.NET Core, processed by middleware.
- **Dependency injection:** A pattern for decoupling class dependencies in ASP.NET Core, where classes receive their dependencies from an external source rather than creating them internally.
- **Singleton:** One instance for the app's lifetime.
- **Scoped:** One instance per request.

- **Transient:** A new instance per service call.
- **Custom middleware:** User-defined middleware for specific functionalities not covered by ASP.NET Core's built-in middleware.

CHAPTER 7

Architectural Principles

Introduction

In this chapter, we will make an overview of some essential aspects of building resilient systems. We begin by outlining the process of constructing a software system, covering key methodologies essential for effective development as architectural patterns and, above all, some of the most insidious antipatterns. Next, we will take a rapid look at how to properly build a software system from scratch to its conclusion: the software life cycle.

After seeing this, we will examine backend architecture layering, discussing the structuring of systems to enhance scalability and maintainability with particular attention to the backend.

We will then focus on functionality-oriented techniques, examining how software functions can be aligned with business objectives to drive strategic development. Finally, we will understand how SOLID principles of object-oriented design integrate with broader architectural strategies, demonstrating their role in improving system robustness.

Through these topics, this chapter provides a concise yet comprehensive framework for understanding and implementing effective software development practices.

Structure

In this chapter, we will cover the following topics:

- Architectural principles and anti-patterns
- Build a software system
- Backend architecture layering
- Functionality oriented techniques
- Connection between SOLID principles and architecture

Objectives

The goals of this chapter are to improve your understanding and skills in designing software from requirements to termination. We will learn how to navigate the complete software development process through thinking, design, and development, ensuring the use of best practices and effective tools and documents. The goal will be achieved through understanding the software life cycle, layering, functionality orientation, and how SOLID principles help us and can properly guide the system architecture.

This comprehensive approach will equip us to design, implement, and manage sophisticated software systems effectively because developing software is essentially solving business problems but, above all, solving people's problems.

Architectural principles and anti-patterns

In this section, we will explore the main guidelines for designing resilient software systems and highlights common pitfalls to avoid. Even though architectural principles are important, one must be much more cautious of anti-patterns because they are insidious. Recognizing anti-patterns prevents recurring mistakes, ensuring the development of efficient, high-quality software that meets long-term business goals. Understanding architectural principles helps us create scalable, maintainable solutions.

Antipattern

An antipattern is a common response to a recurring problem that initially seems to be the best (or a good) solution, but shortly afterward, it turns out to be ineffective and often even counterproductive. While design patterns are best practices that developers follow to solve problems, antipatterns are *pitfalls*.

Antipatterns often arise from a lack of experience, outdated practices or short-term thinking. Recognizing antipatterns is crucial because of the following:

- Lead to code that is inefficient and difficult to understand, maintain, or scale.
- Cause technical debt, where immediate compromises lead to additional rework later.
- Result in processes that stifle productivity and innovation.

Following are some classic anti-patterns in architecture:

- **Big ball of mud:** A software system developed without a clear architecture or design, where the code is disorganized, and dependencies between components are entangled.
- **God object:** Concentrating too many functions in a single part of the software (a single class or module).
- **Golden hammer:** Applying a well-known technology or tool as a solution to every problem, regardless of whether it is appropriate for the current context.
- **Lava flow:** Retaining undesirable (often redundant or obsolete) code because removing it is too risky, or nobody understands it fully.

Big ball of mud

Early in the history of software development, engineers often wrote code on paper. After completing their handwritten code, they would take turns using the limited office computer and compiler to type and verify their work. This method demanded meticulous attention to detail since any error could mean losing a precious turn at the computer/compiler.

In contrast, modern software development practices sometimes

involve writing code without a thoroughly clear plan. This approach can be likened to managing a large plastic bag filled with water. Occasionally, a small hole appears, allowing water to gush out, and a developer patches it with tape. Over time, more holes form, and more patches are applied. Eventually, the entire bag becomes covered in tape, struggling to contain the water. At this point, it often becomes evident that the software needs to be completely rewritten, restarting with a new bag, new water, and new tape.

In fact, this scenario perfectly illustrates what is now known as the **big ball of mud** a term coined in 1997 by *Brian Foote* and *Joseph Yoder* in a paper presented at the *Pattern Languages of Programs* conference. This term describes software systems that lack a perceivable architecture, characterized by a haphazard structure akin to a makeshift patchwork, continually repaired and barely holding together.

A big ball of mud appears following some very specific events:

- Lack of architectural structure
- Rushing to implement something fast and functional without considering all use cases and scenarios
- A small temporary trial software that is always updated with new features added
- Therapeutic persistence on obsolete software trying to keep it alive with old practices and the wrong approach

A common saying in the software industry is as follows:

Nothing is more permanent than temporary software.

God Object

The **God Object** antipattern represents a problematic design approach in software development where a single object or class is overly burdened with a multitude of responsibilities that should ideally be distributed across multiple, smaller, more specialized objects. This design flaw arises when too much functionality is centralized in one place, making the God Object critical to the overall functioning of the application.

In practice, a God Object might handle things like user interactions, data processing, business logic, and database operations, all packed

into a single class. This makes the class bloated and overly complex, which in turn can make the codebase harder to navigate and understand. The large amount of diverse functionality concentrated in the God Object leads to high coupling and low cohesion: the object is tightly linked to many different parts of the program and performs tasks that are not logically related, which is a direct contradiction to fundamental principles of good software design.

This antipattern significantly hampers the maintainability of the software. Since so many components of the system depend on the God Object, making changes to it can have widespread, unintended consequences. This makes debugging and modifying the software risky and time-consuming, as each change has the potential to affect numerous aspects of the system's behavior.

Moreover, the God Object often becomes a major obstacle to unit testing. Since it interacts with many parts of the system and manages various types of data and operations, setting up tests for such an object is particularly complex. The interdependencies require extensive mocking or complex setup of test environments, which goes against the principles of simple and independent test cases, for this refer to [Chapter 12, Automated Testing](#).

Additionally, adhering to the God Object approach violates several object-oriented design principles. The **Single Responsibility Principle (SRP)** advocates that a class should have only one reason to change, suggesting that its functionality should be narrowly aligned with a single aspect of the system. A God Object, by its nature, has multiple reasons to change, which makes the system less flexible and adaptable to new requirements or changes in existing ones.

The emergence of a God Object is often due to expedient but short-sighted decision-making during the initial stages of development, where the ease of adding functionality to a single, already-established object outweighs considerations for long-term system health. This might seem to speed up development initially but invariably leads to a technical debt that grows exponentially as the system scales and evolves.

Golden Hammer

This term comes from the saying, *if all you have is a hammer, everything looks like a nail*, which captures the essence of this antipattern. The **Golden Hammer** antipattern refers to the mindset or approach where a particular technology, tool, or methodology is considered so favourable or familiar that it is used as the solution for every problem, regardless of whether it is appropriate for the task at hand.

In practice, the Golden Hammer manifests when developers or organizations repeatedly apply a known tool or technique in different situations without properly evaluating its suitability. This can happen for several reasons, which are as follows:

- **Familiarity:** Developers or teams may prefer to use tools and techniques they are comfortable with, avoiding the learning curve associated with newer or more appropriate solutions.
- **Success bias:** If a particular tool or approach has been successful in past projects, there is a tendency to overgeneralize its effectiveness, leading to its application in scenarios where it may not perform well.
- **Resource constraints:** Sometimes, due to time or budget constraints, teams might opt to use existing tools or methods instead of investing in alternatives that might be better suited to a specific challenge.

The consequences of the Golden Hammer antipattern can be significant, as follows:

- **Inefficiency:** Using a tool or technique that is not suited to a particular problem can lead to inefficient and suboptimal solutions. This can result in poor performance, higher maintenance costs, and extended development times.
- **Increased technical debt:** Persisting with an inappropriate tool or methodology can complicate the system unnecessarily, adding to the technical debt. Over time, this can make the system more fragile and harder to maintain or upgrade.
- **Stagnation:** Over-reliance on familiar tools and techniques can stifle innovation and prevent teams from exploring more effective solutions that could enhance performance,

scalability, and maintainability.

To counter the Golden Hammer antipattern, it is crucial for teams to cultivate an open-minded culture that values critical evaluation and continuous learning. This involves staying updated with new technologies, regularly reviewing and critiquing the tools in use, and always considering multiple options before deciding on the approach to a problem. By fostering this kind of environment, organizations can ensure that their solutions are not only effective but also appropriately tailored to the challenges they face. From the code perspective, if we design and develop our system with clear boundaries that support the replaceability of individual components, we can easily substitute existing tools and dependencies with more suitable solutions for the problems our software addresses.

Lava flow

This common antipattern in software development, describes a software where parts of the code become obsolete or unclear but are still maintained in the system, often because they are poorly understood or there is fear that modifying them might introduce new bugs. The name **lava flow** evokes the image of lava that solidifies and becomes part of the landscape, difficult to remove or modify.

This antipattern usually occurs in long-running projects, where code is continuously added and rarely restructured or removed. It can also happen when team members change frequently, leaving behind code that no one in the current team fully understands. As a result, the code becomes like layers of solidified rock, with new functionalities built on top of the old, without clear understanding or documentation.

The key features of the lava flow antipattern include:

- **Obsolete code:** Parts of the code that are no longer actively used or might have been superseded by new technologies or approaches and this obsolete code can potentially waste important resources and impacting
- **Mysterious code:** Sections of code that no one in the current team fully understands, often left behind by previous

developers.

- **Difficult maintenance:** The code is so intertwined and poorly documented that modifying it is risky and could cause more problems than it solves.

To avoid the lava flow antipattern in software development, it is crucial to engage in proactive practices such as regular refactoring and comprehensive documentation. Implementing thorough code reviews and maintaining a strong suite of automated tests ensure that the code remains clean and comprehensible. Continuous integration and deployment processes help integrate and test changes frequently, reducing the accumulation of problematic code. Additionally, fostering an environment of ongoing training and mentoring, along with actively removing obsolete or unused code, supports knowledge sharing and prevents the system from becoming encrusted with outdated elements. These strategies collectively are normally called **gardening**, refer to the section *Software gardening* of this chapter, that contributes to a sustainable and maintainable codebase.

Architectural patterns

A software architecture pattern is a general and reusable approach to solving recurring problems that are encountered in the design of a software. Each pattern provides a consistent and repeatable framework for organizing software structures ensuring reliability, adaptability and long-term sustainability, responding to different design requirements. These patterns constitute a guide and should not be used as if they were final projects. They help developers create flexible and efficient systems by providing clear and well-defined methodologies. Architectural patterns help simplify the development process, facilitating collaboration and promoting best practices. They can always be adapted to the specific needs of each project, allowing teams to face challenges while maintaining a proper and organized code base.

In [Chapter 3, Architectures and Core Components](#), we examined the most prevalent architectural patterns foundational to modern software design. This chapter is a comprehensive guide, examining each pattern's structure, typical use cases, benefits, and potential drawbacks. By dissecting these patterns, we aim to equip readers

with a deeper understanding of how various design configurations can be leveraged to build robust, scalable, and efficient software systems.

Architectural principles

Obviously, the fundamental responsibility of an architect is to determine architectural decisions. These decisions often concern the system or application structure and may also include technological choices, especially when they affect architectural traits. In any situation, an effective architectural decision assists development teams in selecting the appropriate technical options. The process of making such decisions requires collecting sufficient pertinent information, rationalizing the choice, recording it, and clearly communicating it to the relevant stakeholders.

Implementing good architectural principles and patterns we develop and provide a framework for making decisions and establishing consistency across projects. They are high-level rules that guide the architectural decision-making process. These principles are typically grounded in the organization's objectives and practices, aiming to align technical strategies with business goals. The most important principles are covered in the following sections.

Modularity

This principle involves breaking down a software system into smaller, manageable modules, each with a well-defined responsibility. Modularity helps isolate system components, making maintenance and testing easier. It also makes it simpler to update or replace parts of the system without affecting the rest of the architecture.

At the heart of modularity is the idea of encapsulation. By encapsulating the details of each module and exposing only necessary elements through interfaces, the system reduces complexity. This separation allows developers to focus on one area of functionality at a time without needing to understand the full details of other areas. The result is that each module can be developed, tested, updated, and understood more easily.

A well-designed modular system is characterized by strong internal

cohesion within the modules and minimal coupling between them. High cohesion means that the internal components of a module are closely aligned with its main function, promoting clarity and purpose. Low coupling, on the other hand, means that modules interact through simple, well-defined interfaces, reducing interdependence. This separation of concerns makes it easier to modify individual parts of the system without affecting others, thus simplifying maintenance and improving scalability. Each module must necessarily correspond to a single functionality, a single task and communicate with other modules in a simple way, ensuring that the overall system becomes more adaptable and manageable. This modular approach not only simplifies the development process, but also improves the long-term sustainability of the software itself, as individual modules can be updated or replaced without the surrounding modules noticing. It also promotes better collaboration between development teams, as clear boundaries and interfaces allow you to work on different modules in parallel.

Another critical aspect of modularity is the independence of modules. By allowing modules to function independently, the system can accommodate parallel development and testing, reducing development time and improving product stability. This independence also supports scalability since individual modules can be enhanced or scaled without necessitating changes to the entire system.

Moreover, modularity inherently supports reusability. A module designed to perform a specific function in one part of a system can often be reused in another context or project without modification. This reusability not only speeds up the development process by reducing the amount of code that needs to be written but also promotes consistency across different parts of the application or across different projects.

The benefits of modularity go far beyond the simple code management. Systems designed with modular principles are generally more reliable and easier to operate. They can adapt more easily to changing user or technology needs, as updates or expansions can be limited to specific parts of the system. Modularity also simplifies debugging and testing, as problems can be quickly isolated within individual modules. This approach not only

increases flexibility and adaptability, but also simplifies maintenance, making the entire system more resilient to change.

Implementing modularity might involve several design patterns and practices, refer to [Chapter 3, Architectures and Core Components](#), for more details, depending on the specific requirements and scale of the system. In some cases, a modular approach can lead to the adoption of architectures like Microservices, where each service is a module that runs independently and communicates over well-defined network protocols.

Overall, modularity in software architecture not only makes a system easier to develop and maintain but also ensures that it can grow and evolve without becoming cumbersome or fragile over time.

Scalability

Scalability is a fundamental concept in software architecture that defines a system's ability to effectively increase its capacity to handle greater amounts of work. This capacity might be needed to manage more data, accommodate more users, or process a higher volume of transactions. The essence of scalability is the system's adaptability to expanding demands without compromising on performance or requiring a complete overhaul of its foundational structure.

When we discuss about scaling a system, we often focus on how well it can grow along two main paths, as follows:

- **Vertical scalability:** This is also known as **scaling up**. It involves boosting the capabilities of existing servers or nodes by adding more powerful processors, more memory, or larger storage components. This approach enhances the ability of the infrastructure to handle increased loads but can sometimes be limited by the maximum capacity of individual hardware components.
- **Horizontal scalability:** This is also known as **scaling out**. On the other hand, it spreads the load across multiple servers or nodes. Instead of making a single system larger and more powerful, horizontal scaling distributes the workload across a network of machines, each handling a

part of the overall load. This method is particularly effective for systems designed around distributed architectures, where tasks are designed to be broken down and processed in parallel.

Achieving scalability is not just about adding more resources. It is about designing systems that can make the most of these resources. This involves strategies such as using load balancers to distribute incoming traffic evenly across servers, ensuring no single server becomes a bottleneck. It also includes designing systems with statelessness in mind, where processes can be easily duplicated and managed without reliance on a shared state.

Moreover, scalability covers how data is handled as the system grows. In large, distributed systems, managing data consistency and efficiency becomes more challenging. Solutions like database sharding (dividing a database into smaller, more manageable pieces, each handled by a different server) can help reduce the burden on any single server and improve overall performance.

Scalability is crucial for maintaining a smooth and responsive user experience as the number of users or volume of data increases. It also offers a cost-effective way to grow, as resources can be added incrementally to match increasing demand without unnecessary expenditure. In competitive markets, the ability to scale effectively can be a significant advantage, allowing businesses to adapt quickly to changing market conditions and user needs.

In a project phase of software or a single function, each architect or developer must pay attention to this crucial aspect of the system lifecycle. Scalability is about building flexible, efficient systems that can grow and adapt without major disruptions or performance degradation, ensuring that as demands increase, the system can continue to perform reliably and efficiently.

Each team member should keep in mind that a system can grow faster than the ability to modify it for proper scaling. In general, it is better to anticipate this possibility rather than modify the software after the first signs that the system is not adequate to handle the required load. Furthermore, in such situations, developers are often under pressure and do not have the time to adequately develop the necessary features, which can have a

negative impact on the quality and testability of the software.

Resilience

Resilience is a critical architectural principle that aims to ensure a system remains operational and reliable under various conditions, including failures and external stresses. This principle involves designing systems that can recover quickly from failures, gracefully handle peak loads, and continue to operate even when parts of the system are compromised.

Resilience in software architecture is fundamentally about designing a system that can handle and recover from various disruptions, ensuring continuous operation and reliability. This concept encompasses the system's ability to anticipate, withstand, recover from, and adapt to unexpected conditions such as hardware malfunctions, spikes in user demand, or software bugs.

A resilient system is constructed with an understanding that components might fail. The goal is to ensure that these failures do not disrupt the overall functionality of the system. For instance, if one server in a distributed network fails, the system is designed to automatically reroute tasks to other servers that are still operational, maintaining the service without noticeable downtime to users.

Resilience also involves the system's capacity to handle a wide range of operational and environmental stresses. This could mean managing sudden increases in web traffic or processing demands without crashing or slowing down significantly. It extends to the system's ability to fend off or mitigate issues arising from these stresses, such as through dynamically allocated resources or automated scaling processes that adjust capacity based on current demand.

The design for resilience is embedded into both the software and hardware components of a system. It includes the use of redundant hardware to avoid single points of failure, as well as software strategies like implementing backups and failovers that ensure data integrity and availability even when parts of the system go down.

Moreover, resilience is not just about reactive measures but also proactive monitoring and maintenance. Systems should be equipped

with monitoring tools that continuously check for any signs of trouble, from slow performance to outright failures, and automated scripts or human operators can take corrective action based on these alerts. This proactive stance helps in minimizing the potential impact of failures by addressing them before they cause significant problems.

A good project phase allows architect and developers building resilience into a system. It means it is rugged enough to handle unexpected challenges and adaptable enough to recover from them swiftly, ensuring that it continues to function effectively and reliably. Thus, it safeguards the user experience and the integrity of the business operations it supports.

Building a software system

Designing a software system from scratch is a complex process that involves multiple phases and different skills. Creating and managing a software system and planning its lifecycle is a multifaceted process that requires careful planning, execution, lifecycle management, and eventual resolution to ensure that it meets its requirements and intended functions effectively throughout its use.

There are many methodologies for approaching software system development, each designed to provide structure, efficiency, and flexibility in different types of projects.

Following are the four widely used methodologies:

- **Waterfall** is a sequential method where each phase must be completed before moving on to the next. The phases include requirements gathering, system design, development, testing, release, and maintenance. It is often used in projects where the requirements are well-defined and stable from the start, such as in industrial or governmental environments. However, it lacks flexibility, any changes made in the later stages require revisiting earlier phases, leading to delays and high costs.
- **Agile** is a methodology that breaks the project into short cycles called sprints, typically lasting one to four weeks. During each sprint, the team develops, tests, and delivers a

working part of the software, allowing for continuous feedback from the customer. Agile emphasizes collaboration between team members and stakeholders, allowing modifications even in the advanced stages of the project. It is suitable for projects where requirements may change frequently, such as digital product development or custom software. Its advantages include flexibility, frequent delivery of features, and reduced long-term risks.

- **Rapid Application Development (RAD)** is based on rapid development cycles with continuous involvement from end users. The goal is to quickly develop functional prototypes of the software, gathering feedback that is immediately used to improve the product. The key phases include planning, design, and iterative development. It is especially suitable for projects with tight deadlines or evolving requirements, such as web applications or business tools. However, RAD may not be appropriate for very complex projects or those requiring high scalability and stability from the early stages.
- Here follows a rapid excursus of the phases of the Waterfall model and what they should include. In this section, we will provide an analysis of the individual phases and offer a mini guide to allow us better understand the topic. Each section will be subdivided in who, what, how and results points able to schematize all the processes.

Definition of requirements

Defining clear and precise requirements is the most important part of any project. In this section, we will give an overview of the process. By effectively understanding and articulating the requirements, we will create the basis for achieving the project objectives, as follows:

- **Who:** Stakeholders and end users
- **What:** To ensure the success of a project, it is crucial to thoroughly understand the initial requirements. It is critical to clearly identify and document key system capabilities and needs based on project objectives and stakeholder expectations.

- **How:** This initial phase involves close interaction with stakeholders to capture all functional and non-functional requirements. Architects, developers, project managers, and clients discuss and document what the software must do and any constraints it must operate within. This stage often uses tools like interviews, surveys, questionnaires, and requirement workshops to ensure all necessary details are collected and understood.
- **Result: Software requirements specification (SRS)** document. The SRS is a formal document that details all the features, requirements, and limitations of a software system. It serves as an agreement between developers and stakeholders on the specifications of the software to be built. It is composed of the following parts:
 - o **Functional requirements:** Details the specific operations that the software must perform.
 - o **Non-functional requirements:** Includes criteria such as performance, security, reliability, usability, compatibility, accessibility and maintainability.
 - o **Interface requirements:** Describes how the software interacts with other systems and devices.
 - o **Database requirements:** Specifies data management and database structure.

The goal is to ensure that stakeholders and the development team have a common understanding of expectations for the software, reducing ambiguity and misunderstanding and providing a basis for testing and evaluating the finished software.

Requirements analysis

After correctly defining the requirements, the natural next step will be to analyse them in depth, as follows:

- **Who:** System analysts
- **What:** Analysing the requirements to ensure they are clear, complete, and achievable is crucial. During this process, it is important to identify any contradictions or ambiguities that

could hinder the project's progress and address them promptly to maintain the integrity and feasibility of the project goals.

- **How:** Armed with a comprehensive set of requirements, the team analyses this information to define system specifications. This phase also includes planning the overall project scope, determining resources (such as personnel and technology), and setting a realistic timeline. Risk assessments are conducted to identify potential issues early in the project lifecycle.
- **Result:** SRS document of requirements validated and approved by stakeholders.

Decisions and trade-offs

Each project involves a series of crucial decisions and inevitable compromises. In this section, we will have an overview of the art of making informed choices that balance competing priorities.

- **Who:** Architects and their development teams
- **What:** Every day, an architect faces the reality that every decision carries consequences. In the definition phase of software development, architects and their teams establish patterns and strategies rooted in fundamental architectural principles. This approach generally serves well until a scenario arises where two potential solutions for the same problem present themselves, both seemingly viable.
- **How:** At this juncture, the decision becomes critical. The team, led by the architect, must choose one of the two options. The selected solution will inevitably influence the software's development trajectory and the speed at which the necessary features are implemented to meet the project's objectives. This choice underscores the importance of thoughtful decision-making in architecture, as it directly impacts the efficiency and effectiveness of the development process.
- **Result:** Architecture decision records (refer to the next section about ADRs) documents. The choice made influences the software's development trajectory, affecting the speed

and manner in which necessary features are implemented to meet project objectives.

Making decisions by ADRs

When a software architect or a team makes an architectural decision, the best way to trace a decision and its consequences is through a methodology called **architecture decision records (ADRs)**. ADRs are a set of documents that capture critical architectural decisions made during development of a system (software design), along with their context and consequences. These records serve as a vital part of the documentation in software projects, providing clear explanations for why certain architectural paths were chosen over others.

Purpose and content

ADRs document the rationale behind each decision, detailing the problem, the considered alternatives, the reasoning for choosing one alternative over the others, and the implications of the decision. This approach ensures that the decision-making process is transparent and understandable, even long after the decisions are made.

Structure of ADR

Typically, an ADR is a simple text file that includes several key sections:

- **Title:** Clearly states what the decision is about.
- **Status:** Indicates whether the decision is proposed, accepted, rejected, or deprecated.
- **Context:** Describes the issue that needs resolution and the conditions at the time of the decision.
- **Decision:** Specifies what was decided and why.
- **Consequences:** Discusses the results of the decision, including benefits and potential downsides.

Benefits of ADR

There are several benefits to adopting this technique during the

lifecycle of our software product. The most important ones are as follows:

- **Improved communication:** ADRs provide a historical context for why decisions were made, aiding new team members in understanding the project's architectural evolution.
- **Facilitation of decision-making:** By forcing the articulation of decisions and their reasoning, ADRs help ensure that choices are thought through systematically.
- **Record for future reference:** They serve as a reference that can help prevent revisiting the same decisions repeatedly or can provide a basis for revising decisions as project conditions change.

Usage of ADR

ADRs are particularly useful in large projects with multiple collaborators, as they ensure all stakeholders are aligned on the project's architecture and can refer back to the documented decisions at any point. They are also valuable in smaller projects to help maintain a clear, long-term vision of the project's architectural guidelines and choices. Overall, ADRs are a pragmatic way to document and manage the key decisions that affect the structure and functioning of software, ensuring that these decisions are deliberate, well-founded, and accessible to everyone involved in the project.

System design

System design is where abstract needs and ideas are transformed into concrete solutions. In this section, we will explore broadly the process of creating a system architecture so that it can be robust and efficient that meets customer requirements.

Refer to the following:

- **Who:** Software architects
- **What:** Design the high-level architecture of the system by defining modules, components, and their interactions, creating diagrams like **Unified Modelling Language (UML)** to visualize the architecture and data flow. This

comprehensive approach facilitates a clear understanding of the system's structure and promotes effective communication among team members.

- **How:** With the requirements set and the system analyzed, the design phase translates this information into a software architecture. This step involves making critical decisions about the high-level structure of the system and the design of individual components. This might include the choice of programming languages, the database design, the data flow diagrams, use case diagrams, and the overall system architecture, including APIs and integration points. A well-known technique for describing the system is the C4Model (<https://c4model.com/>).
- **Result: Software architecture document (SAD)** document, this second document is also crucial for the definition of a software system and is the root of the software development process. Understanding how to write this document is fundamental for each software architect.

The main parts of the SAD document are the following:

- **Architecture description**
 - **Architectural views:** Presents different architectural views (e.g., logical, physical, and development) to explain components and interactions.
 - **Main components:** Describes the main modules of the system, their responsibilities, and interactions.
 - **Technologies used:** List of technologies and tools used.
- **Reasons for the architectural choices**
 - **ADRs:** As seen in the previous paragraph, *Making decisions by ADRs*, this is the right place to put the whole document.
 - **Impact:** Discussion of the impact of decisions on performance, maintainability, and other aspects of the system.

The SAD serves to orient the development team, ensuring that the architecture is aligned with the requirements and objectives of the

project. It is essential for design consistency during software implementation, maintenance, and evolution.

Implementation of system design

In this section, we will explore the steps and strategies needed to transform the project from paper specifications to reality.

Refer to the following:

- **Who:** Development team
- **What:** Code the software based on the technical specifications (SAD) based on specifications (SRS).
- **How:** During implementation, developers write code according to the previously established design specifications. This phase often sees teams divided into smaller groups focusing on various aspects of the system, such as the user interface, business logic, database management, and more. Effective coding standards and peer review practices are crucial here to maintain code quality and consistency. In the late 1960s, *Melvin Conway* observed a phenomenon now known as **Conway's Law**: organizations that design systems are inevitably shaped to create designs that mirror the communication structures within these organizations. Remembering this law, the architect should build the developer teams accordingly.
- **Result:** Working software.

Testing of software

Software testing is the process of evaluating and verifying that a system meets the specified requirements. For more details about this subject, refer to [Chapter 12, Automated Testing](#), and [Chapter 19, Debugging, Testing, and Performance Tuning](#).

- **Who:** Testers
- **What:** Test software to discover and fix bugs.
- **How:** Use specific tools or frameworks for each type of test. This process verifies that each component, functions and requirements are respected. We can do that through the following five types of tests:

- o **Unit tests:** Evaluate the functionality of specific sections of code, typically functions or methods, to ensure they perform as intended in isolation.
- o **Integration tests:** Check the interaction between different modules or services in a software system to ensure they work together as expected.
- o **System tests:** Assess the complete and integrated software system's compliance with specified requirements, examining the full application's functionality and performance.
- o **End-to-end tests:** These are positioned similarly to system tests but with an even broader perspective. While system tests focus on the software's compliance with technical specifications and functional requirements, E2E tests verify the overall user experience from start to finish.
- o **Acceptance tests:** Involve testing the software with the intention of determining whether it meets the business needs and is acceptable for delivery to the end-users.

Each type of test is designed to validate a precise domain of software.

- **Result:** Verified and validated software.

Deployment of software

The deployment of a software involves several stages including release, installation, configuration and activation. All of this serves a software application to work in a specific environment or production.

- **Who:** System engineers
- **What:** Deploy the software in a production environment, then continuously monitor system performance and stability to ensure optimal operation.
- **How:** After testing, the software is ready to be deployed into a production environment, where users can begin to operate it. This phase may involve initial deployment to a limited

audience to gauge performance and collect user feedback.

- **Result:** Software system and accessible to users.

Post-deployment care

Now that our software has been released into production and so the customer is using it, as a company we have to keep the software produced. We will probably pass it on to the support team, which can behave in two different ways depending on the organization of the company. You can either maintain the software or you can take care of it and grow it as if it were a garden.

Software maintenance

Software maintenance is the process of modifying, updating and improving software applications after their initial release. This includes fixing bugs, improving features, optimizing performance, and ensuring compatibility with new hardware or software environments. The goal of this phase is to keep the software operating as it is efficiently and effectively throughout its life cycle, ensuring that it continues to meet the needs of users and works smoothly.

Refer to the following:

- **Who:** Maintenance team
- **What:** Address and resolve emerging problems, enhance the system with updates to improve performance, and incorporate new features based on user feedback to better meet user needs.
- **How:** Post-deployment, the software needs ongoing maintenance to correct any issues, improve functionality, and enhance performance based on user feedback. Regular updates are also necessary to adapt to changes in the operating environment or to add new features as the user needs to evolve.
- **Result:** Updated and optimized system.

Software gardening

Software gardening is an approach to software maintenance that

emphasizes the continuous improvement, improvement and growth of the software system. This is a metaphor to describe an approach to programming that emphasizes organic growth and continuous improvement of code over time. In software gardening, the development of software is seen more like tending to and maintaining a garden. This approach focuses on proactive care, including code refactoring, enhancing functionality, and adapting to new technologies to keep the software healthy and robust.

Refer to the following:

- **Who:** Maintenance and gardening team
- **What:** Software development, just like gardening, requires regular code pruning and maintenance to improve readability and performance, adapting to new technologies and user demands in response to changing conditions, and recognizing that, similar to plants, the software matures and evolves through continuous iterations.
- **How:** The underlying idea is to treat software not as a finished product but as a living organism that requires ongoing attention and regular modifications to adapt and thrive in its environment. This approach promotes agile development practices, like **extreme programming (XP)**, and similar methods that emphasize flexibility, collaboration, and adaptation.
- **Result:** More, resilient, adaptable, and continuously improving software system.

Definition of termination criteria

Termination criteria are default conditions that determine when the entire development, maintenance and support process should be completed. These criteria ensure that all necessary tasks have been completed, that the requirements have been met and that the software is declared dead.

Refer to the following:

- **Who:** Stakeholders, end users, and support teams
- **What:** Considering the conditions under which software should be terminated, such as obsolescence, financial non-

viability, and technological redundancy, it is essential to assess the ongoing costs versus the benefits of maintaining the software and evaluate the impact of its termination on users and stakeholders.

- **How:** This final phase involves careful consideration and consultation with stakeholders to decide when and how the software should be terminated. It includes evaluating the software's relevance, usability, and financial sustainability. Tools such as feedback forms, user analytics, and financial reports might be used to gather relevant data. Discussions and meetings are held to ensure that all parties understand the implications and are prepared for the software's end of life.
- **Result:** The **Software Termination Document (STD)** is a formal document that outlines the decision to end the software's lifecycle. It serves as a mutual agreement between developers and stakeholders on how the termination process will be managed. It is composed of the following parts:
 - **End-of-life announcement:** Includes the official notification of software termination to users and stakeholders.
 - **Migration path:** Provides details on transitioning to alternative solutions or upgrades.
 - **Data preservation:** Outlines methods for data extraction, storage, and migration to ensure data integrity and accessibility post-termination.
 - **Timeline:** Specifies key dates in the termination process, including final updates, support withdrawal, and complete shutdown.

The goal is to ensure a smooth transition for all parties involved, minimizing disruption and maintaining data integrity and accessibility, ultimately leading to a clear and well-managed end of the software's lifecycle.

By navigating the stages of software lifecycle management effectively, architects and developers can build a software system that is solid, scalable, and aligned with the strategic objectives of

the organization or client. Each step, from defining requirements to planning for the software's end of life, requires diligent attention to detail and coordination among all team members to ensure the final product is of high quality, meets all expectations, and has a clearly defined and well-managed lifecycle. This comprehensive approach helps ensure that the software remains effective throughout its intended lifespan and that its discontinuation is handled smoothly and systematically. Whether the software is small or very large, strong planning produces excellent software.

Backend architecture layering

Backend architecture layering is one of the most used concepts in the development of scalable and maintainable software applications. It is also known as **n-tier** architecture. This type of architecture is easy to implement and easy to maintain. The easiest layering is as follows:

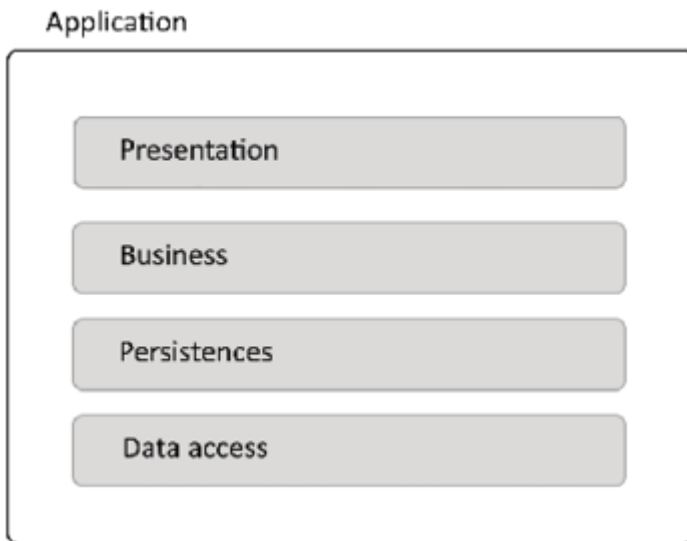


Figure 7.1: Simple layering application

This approach involves organizing code into distinct layers, each with a specific responsibility, even if the deployment will be as one. This separation of concerns ensures that the backend in ASP.NET Core is organized into three parts: business logic, persistence, and

data access layer (DAL).

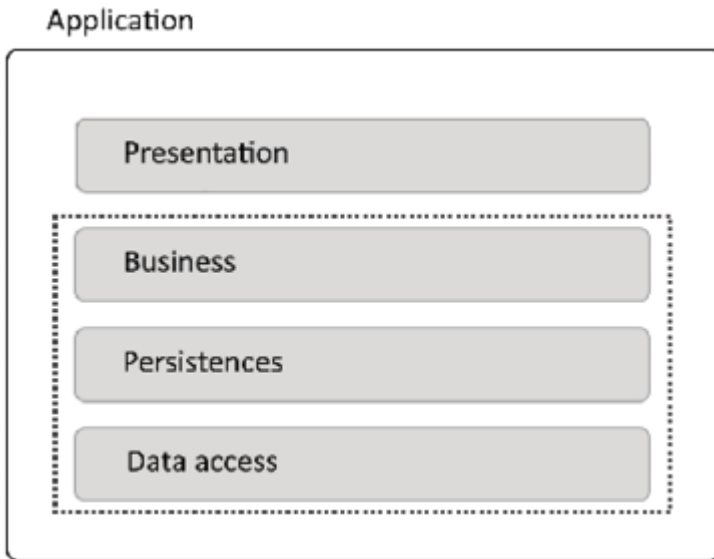


Figure 7.2: Backend layering in the application

Each layer interacts with others through well-defined interfaces, allowing for more manageable development and maintenance.

The essence of layering

As we said, the above simple layering application is a single deployment too, that we can see as a single executable. If we do not have an iron discipline, we can easily take shortcuts that break the harmony of the layers, causing our splendid code to slowly slide into a **big ball of mud** accumulating, day by day, a very big list of technical debts impossible to fix. After a while, we can find business logic in the presentation layer and vice versa (only for example).

There is a technique that can avoid this problem by design. Refer to the following figure:

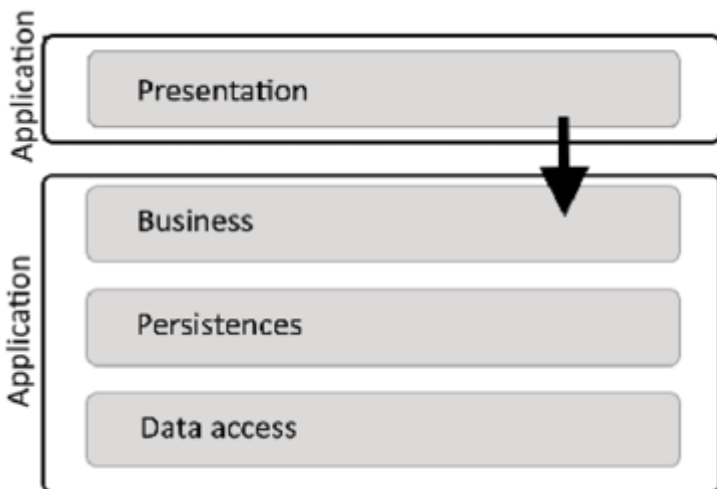


Figure 7.3: Separation in two applications

The separation of concerns here is one step beyond. There are two applications that communicate in an out-of-process way.

In the scenario described above, where two applications communicate in an out-of-process way, this principle is extended by having the applications operate independently while communicating through **inter-process communication (IPC)**. IPC allows them to exchange data across different processes, with another initially unexpected advantage: the two applications can be run on different machines as well as on the same machine.

This architecture enables each application to be developed, deployed, and scaled independently, which is beneficial for system resilience and adaptability. In an application ASP.NET Core, two applications could be the front end (i.e. with Blazor) and the backend with the layering above.

But the backend (formed by business, persistence and data access layers) can suffer from the subdivision above. Then, in the extreme, we can imagine a separation of each layer in separate applications or, even more, each domain module that communicates with others via socket or something like that. The layering figure will look like as follows:

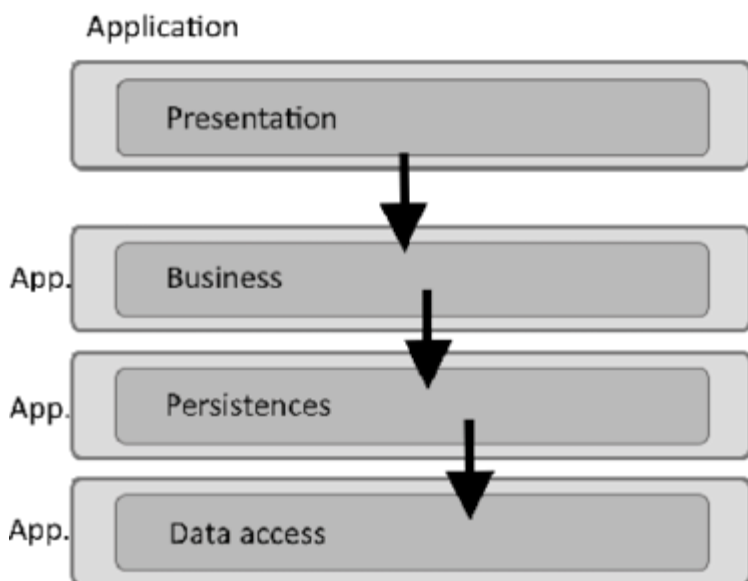


Figure 7.4: Separation in more applications

Layering the backend architecture

In backend architecture, the most fundamental rule is the **Dependency Rule**: source code dependencies can only point to the bottom (as shown previously in *Figures 7.2, 7.3, and 7.4*). Nothing in the lower layers can know anything at all about something in the higher layer. This rule ensures that the lower layers are isolated from the changes in the higher layers.

We have seen in [Chapter 3, Architectures and Core Components](#), the Clean Architecture that has no flat layers but circular layers. For reference, [Figure 7.5](#) is the principal diagram of Clean Architecture:

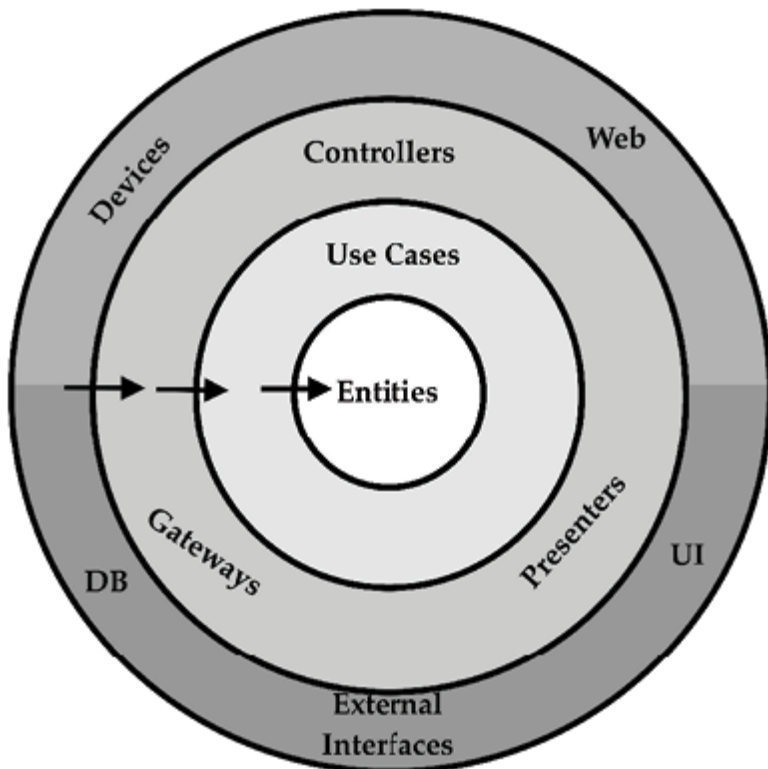


Figure 7.5: Clean Architecture

According to the layering architecture, there is another constraint: the flow is from outer to inner, which is exactly how layering is from top to bottom. The *entities* layer forms the core of the application. This layer should consist of simple data structures and business rules that are critical to the functionality being provided by the application. These entities should be plain objects without any business logic related to database operations, web frameworks, or any external element.

Clean Architecture as separated applications

An application that follows Clean Architecture principles could also be separated into multiple small targeted applications, as we can see in the following figure:

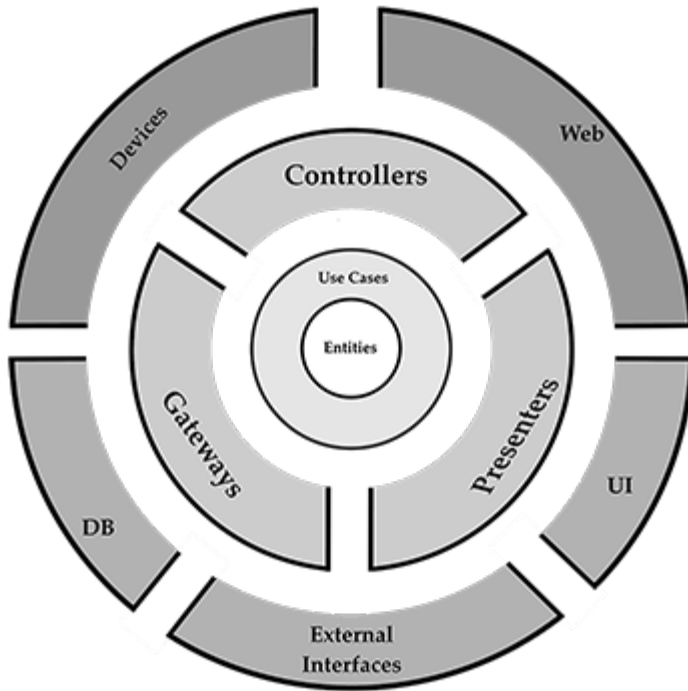


Figure 7.6: Clean Architecture divided by applications

The use-cases layer contains the application-specific business rules. It interacts directly with entities to enforce the application's business rules and policies. The **Use Cases** layer orchestrates the flow of data to the **Entities** and ensures that the output is executed in a manner that the external layers can utilize. These two circles, representing **Entities** and **Use Cases**, could belong to a single application that exchanges all information with others via a socket. In our ASP.NET Core application, we can figure out an application that communicates with others using SignalR, as detailed in [Chapter 11, SignalR in Real-time Applications](#). Before interpreting requests and sending responses to and from each application, we may need **interface adapters** (as we have seen in *Hexagonal Architecture* in [Chapter 3, Architectures and Core Components](#)) that convert data between the most convenient forms of use cases/entities and external layers.

The other two circles are subdivided into more applications according to the domain model. In this case too, the communication way is from outer layers to inner layers. In this scenario, a new constraint arises: determining which specific communications should be enabled between different modules. This involves defining the appropriate channels and protocols for interaction among various software architecture components. Properly addressing this constraint is crucial for ensuring that each module can exchange information effectively without compromising the system's overall functionality and security. Establishing these communication pathways facilitates the integration of distinct modules, allowing them to function cohesively while maintaining the principles of modularity and separation of concerns. This decision impacts not only the technical implementation but also the scalability, performance, and maintainability of the application.

Benefits of layered architecture

Layering the backend as per flat layers, as per Clean Architecture, as well as in any other paradigm we want offers several benefits, as follows:

- **Independence from frameworks:** The system becomes easy to upgrade or replace components without affecting the core business logic or other modules. For instance, changes in the business logic affect only the use-cases or entities, not the entire application.
- **Testability:** With separation by layer, it becomes easier to implement unit tests.
- **Scalability:** Independent layers mean that you can scale out specific areas of the application without reworking others.

Implementing Clean Architecture as layered architecture involves careful consideration and meticulous layering of components to ensure each piece operates independently yet seamlessly together. By adhering to the **Dependency Rule** and organizing code into clear, defined layers, we can create systems that are hardy, flexible, and maintainable. This approach not only makes the development process clearer but also makes the system more adaptable to new business needs.

Functionality oriented techniques

In the landscape of modern software development, adopting functionality-oriented techniques is crucial for ensuring efficient and maintainable systems. Among these techniques, **slicing** and **communication optimization** play a key role, especially in contexts where the backend must effectively interact with a variety of devices through the network. Developing a software system focusing on a single functionality allows us to analyze and develop it to the best without external distractions due to the other functionalities of the system.

Vertical slicing

The concept of vertical slicing refers to the practice of dividing application functionality into slices that cut through all layers of the software architecture, from the use cases/entities to the user interface. This approach contrasts with the more traditional **horizontal layering** where components are organized into separate functional layers. In backend, vertical slices are particularly useful for isolating specific domain logic and for promoting a service-oriented or microservices architecture. Each piece works independently, allowing modular development and distribution, i.e. the possibility to add or remove a particular functionality in a simple way and without the rest of the system noticing or suffering. This type of modularity facilitates both maintenance and scalability, as changes are limited to individual slices so without affecting the entire system. Vertical cutting improves team collaboration by allowing parallel development on different system functions, reducing dependencies and obstacles. Adoption of this approach can lead to more agile and responsive development processes, as new features and updates can be delivered gradually. This approach is also well aligned with agile methodologies that support **continuous integration and continuous delivery (CI/CD)** practices. By focusing on end-to-end functionality, vertical functionality ensures that all components work together smoothly, improving overall system cohesion and reliability. This approach can be useful in large and complex applications, where managing dependencies and inter-layer interactions can become burdensome. It also helps to optimize performance, as each slice can be adjusted and resized

independently according to its specific needs. This method supports the evolution of the system allowing for gradual refactoring and the integration of new technologies without interrupting existing functionality. This approach promotes a culture of continuous improvement, enabling teams to adapt to changing business needs and technological advances. This method represents a shift towards a more flexible and durable architectural style, offering numerous advantages in terms of development efficiency, system maintenance and overall agility.

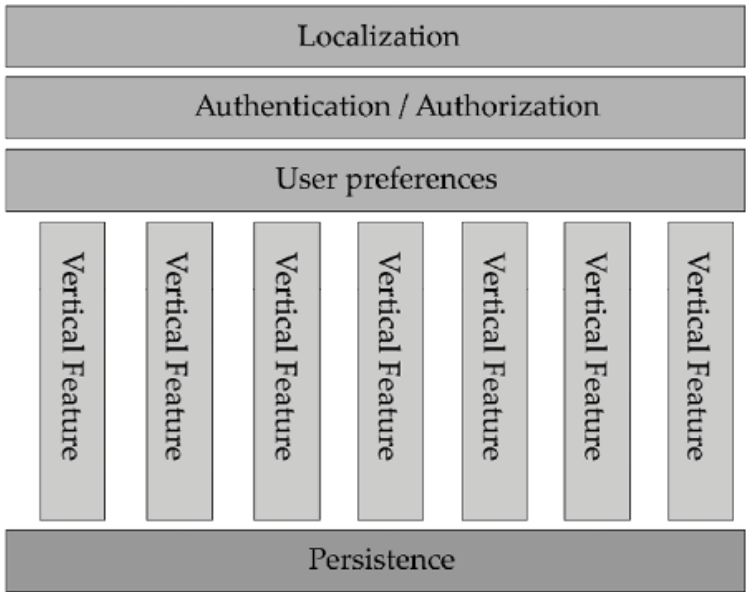


Figure 7.7: Vertical slicing

In the case of [Figure 7.7](#), we have some **horizontal layering** interpreted by the **Localization layer**, **Authentication/Authorization layer**, **User preferences** and **Persistence layer** because these horizontal layers act in a transversal way throughout the entire application.

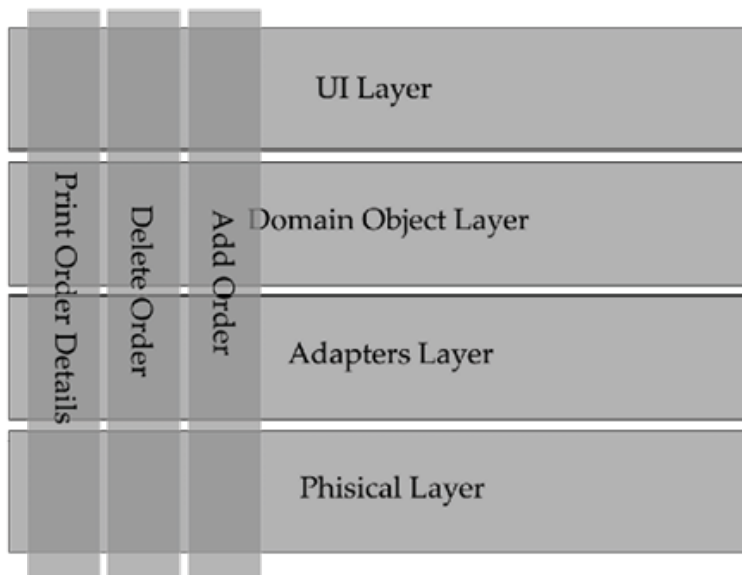


Figure 7.8: Vertical slicing example

Another way to see this concept is represented in [Figure 7.8](#). A traditional layered architecture with vertical subdivision is represented here for specific operations such as add order or delete order. These operations begin at the UI layer and extend directly to the physical layer, which brings together essential resources such as databases and printer drivers. This design promotes the separation of concerns by structuring the application into distinct layers, each of which handles a specific aspect of functionality, from user interactions to direct hardware management. This approach simplifies maintenance and improves system scalability.

Feature vertical slicing

In the previous paragraph, we have seen the **Vertical slicing** theory. In particular, we can see how a single vertical feature should be implemented.

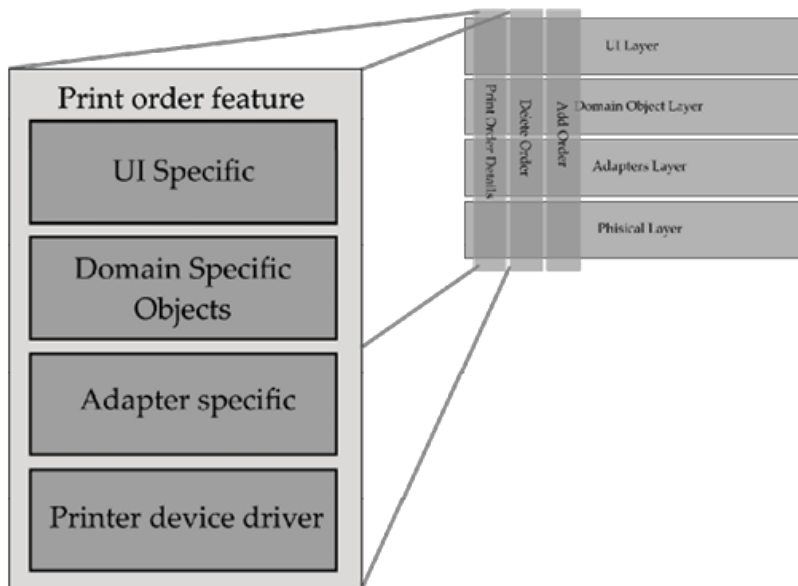


Figure 7.9: Vertical slicing in depth

In [Figure 7.9](#), we can see how to structure the layering of a vertical feature (in this case, the print of an income order) with a layering technique. We have his specific UI, specific domain objects, specific UI adapters, and finally, the printer driver. Here, we have the same structure of layers of the main application, but we could have some other architectural structure. Even more, we can decide to have an external process to be integrated into the main application or simply a microservice that provides the functionality.

Connection between SOLID principles and architecture

As we have seen in [Chapter 1, Introduction to ASP.NET Core](#), the SOLID principles are fundamental guidelines in object-oriented software development that help create cleaner, more flexible, and reliable code. These principles help every developer reduce dependencies, improve code modularity, and make their systems easier to extend and modify without breaking existing functionality. Even if SOLID is the only principle of good coding structure, there is a strong relationship between the SOLID principles and software architecture, emphasizing practical implementations in system design.

Following is how these principles integrate with and support software architecture:

- **Code structuring and organization:** SOLID principles guide software architects in defining how to organize and structure code so that different components of the system are easy to manage, test, and develop independently. For example, the SRP ensures that each module or class has only one reason to change, thus avoiding excessive complexity and dependencies.
- **Scalability and maintainability:** By following the **Open/Closed Principle**, the architecture becomes more flexible and open to extension but closed to direct modification. This allows architects to design systems that can be extended with new features without altering existing code, thus facilitating updates and scalability without disrupting already tested and functioning components.
- **Reduction of technical debt:** Applying principles like **Liskov Substitution** and **Interface Segregation** helps reduce technical debt by ensuring that interfaces are well-defined and that derived classes can replace their base classes without side effects. This helps keep the architecture clean and prevents future issues during maintenance or expansion of the system.
- **Testability:** A system designed following SOLID principles is generally easier to test. For example, the **Dependency Inversion Principle** promotes a design where high-level dependencies do not depend on low-level ones, but both depend on abstractions. This not only makes the system less sensitive to changes in specific components but also facilitates the substitution of components during testing (e.g., with mock objects).
- **Integration and deployment:** With a modular and flexible structure, the integration of new components or external systems becomes more manageable. An architecture that adheres to SOLID principles can facilitate continuous deployment and continuous integration processes, reducing the risks of errors and downtime.

SOLID principles are not just rules for writing clean code. They are fundamental to software architecture as they guide the creation of systems that are sustainable in the long term. They help architects make decisions that improve the quality of the software, making the system more flexible, maintainable, and adaptable to future needs.

Conclusion

The role of a software architect is complex, requiring significant responsibility and expertise. Key aspects include making impactful decisions about architectural design, scalability, and long-term maintainability, which affect the project's trajectory and team morale. Staying updated with evolving technologies and critically assessing new innovations for integration is crucial. Understanding the entire software lifecycle ensures robust, adaptable system design. Despite challenges, the role is highly satisfying and creative, offering the freedom to innovate and turn abstract concepts into real-world solutions, while also solving complex problems and mentoring others.

In the next chapter, we will explore CQRS, discussing its principles, benefits, and practical implementation. We will learn how to design systems that separate read and write operations to enhance performance and scalability.

Points to remember

- **Architectural principles:** Guidelines that dictate the overall structure and management of a system, ensuring it is scalable, maintainable, and aligned with business goals. Architectural principles are modularity, scalability and resilience.
- **Software antipattern:** A common response to a recurring problem that is usually ineffective and risks being counterproductive to software development practices.
- **N-tier architecture:** Backend architecture layering, also known as n-tier architecture, is a widely used approach that facilitates the development of scalable and maintainable software applications through its ease of implementation

and maintenance.

- **Vertical slicing:** Divide a project into smaller, manageable pieces that each deliver a complete feature across all the necessary layers from the user interface to the backend databases or physical layer.

Multiple choice questions

1. What does a software architect do?

- a. Translates business requirements into technical solutions.
- b. Manages human resources within the development team.
- c. Performs manual testing on code.
- d. Designs graphics for the user interface.

2. Why is backend layering important?

- a. To increase the system's complexity.
- b. To decrease the system's scalability.
- c. To improve maintainability and scalability.
- d. To minimize stakeholder involvement.

3. What is the purpose of functionality-oriented techniques in software?

- a. To increase the project budget.
- b. To align software functions with business objectives.
- c. To decrease communication among teams.
- d. To complicate development processes.

4. What is a software antipattern?

- a. An ideal model that everyone should follow.
- b. An effective solution to common problems.
- c. An ineffective practice that can be counterproductive.
- d. A new and cutting-edge technology.

5. What role do SOLID principles play in software development?

- a. They complicate the development process unnecessarily.
- b. They help in making the user interface more attractive.
- c. They enhance system robustness by promoting best practices in object-oriented design.
- d. They are mainly used for database management.

6. What is the primary purpose of an SRS document?

- a. To provide a detailed codebase for developers to start coding immediately.
- b. To outline the software's architecture and design principles.
- c. To detail the software's requirements, ensuring stakeholder agreement.
- d. To serve as a marketing tool for attracting investors.

Answer key

Key terms

- **Software architect:** A professional who designs the high-level structure of software systems, ensuring alignment with business and technical requirements.
- **Backend layering:** The practice of structuring server-side software into distinct layers to improve scalability and maintainability.
- **Functionality-oriented techniques:** Approaches in software development that focus on aligning software functions with business needs to drive effective solutions.
- **Software antipattern:** A commonly used software development practice that is initially perceived as a good solution but is actually ineffective and counterproductive.
- **Software requirements specification document:** A detailed description of software from the point of view of the requirements.
- **Software architecture document:** A comprehensive

document that describes the system architecture of a software project, detailing its components, relationships, and properties.

- **Architectural decision records:** Records that capture the essential details of why a particular decision was made regarding the software architecture, along with its context and consequences.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 8

GoF Design Patterns

Introduction

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides are known as the **Gang of Four (GoF)** because they co-authored the book *Design Patterns: Elements of Reusable Object-Oriented Software*, published in 1994. The term *GoF* was adopted by the software development community to refer to them as a cohesive group that had a significant impact on the field of software engineering, particularly for their work on identifying and cataloging design patterns.

This chapter will examine the key aspects of GoF design patterns and their relevance to ASP.NET Core development. We will start by studying six of these patterns because they are important in the ASP.NET Core applications. We will then apply this knowledge practically, starting with implementing creational, Singleton, and Factory Method design patterns while looking at how they should be used to optimize resource management and object creation. We will look at the Decorator Pattern to improve the functionalities of a class without touching the original code. We will then study the use of the Observer and Strategy behavioral models to improve event management and data processing dynamics within ASP.NET Core

applications. The chapter will also highlight best practices and common pitfalls in implementing these patterns in ASP.NET Core, providing guidelines that ensure effective use while avoiding common mistakes.

Structure

In this chapter, we will cover the following topics:

- Application of GoF design patterns in ASP.NET Core
- Singleton Pattern in ASP.NET Core application
- Entity Framework Core overview
- Factory Method Pattern with EF Core
- Decorator Pattern for logging
- Observer Pattern for application state
- Strategy Pattern for managing payments

Objectives

By the end of this chapter, readers will have an understanding of how to implement the GoF design patterns within ASP.NET Core applications. We will explore each pattern's role and benefits to see how they can enhance our code's reusability, maintainability, and scalability. We will learn how to effectively apply patterns such as Singleton, Factory Method, Decorator, Observer, and Strategy to tackle common design challenges and optimize the architectural framework of our ASP.NET Core projects. One point of digression is a paragraph to explain how Entity Framework Core works and why it is useful.

Application of GoF design patterns in ASP.NET Core

The most common definition of design pattern is, *a reusable solution to a common problem encountered in daily software development*, but the definition we can find in the book of the GoF is as follows:

Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times

over, without ever doing it the same way twice.

In their book, they divided the design patterns into three main groups, as follows:

- **Creational patterns:** They simplify the object creation process, shielding a system from the details of how objects are made, composed, and represented. This abstraction promotes flexibility and modularity, making the system less dependent on the concrete classes of objects created.
- **Structural patterns:** They facilitate the arrangement of classes and objects into larger structures, promoting flexibility and increased functionality. They help manage complex compositions by ensuring the system remains scalable and easy to understand, which is crucial for building large-scale applications.
- **Behavioral patterns:** They are essential for managing complex communications and responsibilities among objects in a system. They focus on improving the interaction and control flow between objects, allowing for more dynamic and flexible handling of operations and changes in the software's behavior.

Several design patterns exist in the aforementioned categories, such as the Factory Pattern or Singleton. In this book, we will not cover all of them, but we will apply them to solve specific problems, starting from the definition of the problem.

Singleton Pattern in ASP.NET Core application

According to the GoF categorization, the Singleton design pattern falls under the category of creational patterns, ensuring that a specific class maintains a sole instance throughout the application. It establishes a unique, globally accessible point for this instance, facilitating controlled access and centralized resource management.

Concepts

This pattern is valuable in contexts where uniform access to resources is crucial, such as managing a common resource in networked systems or sustaining a unified configuration object

across an application. By limiting a class to only one object, the Singleton Pattern effectively minimizes resource consumption, leading to a more efficient and simplified architectural design.

How it works

This design pattern ensures that only one instance of a particular class is created throughout the lifecycle of an application. The pattern provides a global access point to that instance, making it easily accessible from different parts of the application without creating multiple copies of it. The uniqueness of this instance is managed internally by the class itself.

When the need arises to use this class, the application will refer to this single instance rather than creating a new one. This ensures that all parts of the application share and use the same instance, maintaining consistent state and behavior across the system.

Implementation of Singleton Pattern

There are several methods to implement the Singleton Pattern. The simplest one is to implement a class with a static member that manages the unique instance of the class itself. However, there are some issues to consider. One of them is that access to a static variable might not be thread-safe, which can be a significant problem in a web application. An example of a Singleton implementation might be the following:

```
1. public class MyClass
2. {
3.     private MyClass() { }
4.
5.     private static MyClass instance = null;
6.     public static MyClass Instance
7.     {
8.         get
9.         {
10.             if (instance is null)
11.                 instance = new MyClass();
12.             return instance;
13.         }
14. }
```

```
14.     }  
15. }
```

As mentioned before, this class is not thread-safe because two instances could have access to the test **if (instance is null)** in the meantime, and if they find it to be true, then each thread can create a new instance. This behavior defeats our minimal efforts to create a single instance by driving us crazy to find the problem. Due to this behavior, this design pattern is sometimes regarded as an antipattern by developers.

A good solution to avoid such problems is as follows:

```
1. public class Singleton  
2. {  
3.     // Lazy initialization of the singleton instance.  
4.     // The Lazy<T> constructor guarantees  
5.     // thread safety in instance creation.  
6.     private static readonly Lazy <Singleton> lazy =  
7.         new Lazy<Singleton>(() => new Singleton());  
8.     // The instance property for accessing the singleton instance.  
9.     public static Singleton Instance => lazy.Value;  
10.    // Private constructor ensures no external instantiation.  
11.    private Singleton()  
12.    => Console.WriteLine("Singleton instance has been  
    created.");  
13.    // Any singleton methods or properties can be added below  
14.    public void DoWork()  
15.    => Console.WriteLine("Doing work...");  
16. }
```

This code offers a simple and efficient way to implement a thread-safe Singleton with lazy initialization in C#. It leverages the features of the .NET to handle the intricacies of lazy instantiation and thread safety.

In ASP.NET Core, Singleton instances are registered with the **dependency injection (DI)** engine and are declared at the application startup. They are instantiated the first time they are requested as they are in a lazy loading initialization, as seen in the previous example. Once created, a Singleton service remains available, and its instance is reused across the application for the

lifetime of the application. This ensures that the same service instance is consistently used throughout the application, providing a shared resource or functionality wherever needed. From the ASP.NET Core application point of view, the Singleton classes are used independently for all connected clients, but class instance remains the same. For example, a good Singleton instance could be a useful class that provides configuration values or standard conversions like from and to Base64 or Celsius to Fahrenheit.

For the purpose of this book, we can create the following class only to share a property value. This will be a normal class without any specific precautions as follows:

```
1. public class MyClass
2. {
3.     public int MyProperty { get; set; } = 123;
4. }
```

Now, we can let the ASP.NET framework do the work. After creating a new project (for instance Blazor WebApp), open the **Program.cs** and before the line **var app = builder.Build();** add the following line:

```
1. builder.Services.AddSingleton<MyClass>();
```

With this, the ASP.NET Core dependency injection engine will be responsible for our Singleton, and our class will be available in the whole application.

We have encountered the service lifetime Singleton in [Chapter 6, Middleware and Extensibility](#). This lifetime modifier in ASP.NET Core is translated as **AddSingleton**. We can try to understand how it works in depth. If we go to the ASP.NET core GitHub repository, we can find the static class **ServiceCollectionServiceExtensions** that implements the **AddSingleton** as follows:

```
1. public static IServiceCollection AddSingleton(
2.     this IServiceCollection services,
3.     Type serviceType,
4.     [DynamicallyAccessedMembers
5.         (DynamicallyAccessedMemberTypes.PublicConstructors)]
6.         Type implementationType)
7. {
```



```
8.     ThrowHelper.ThrowIfNull/services);
9.     ThrowHelper.ThrowIfNull(serviceType);
10.    ThrowHelper.ThrowIfNull(implementationType);
11.
12.    return Add(services, serviceType
13.               , implementationType
14.               , ServiceLifetime.Singleton);
15. }
```

In the aforementioned code, the **ServiceDescriptor** is a class defined as, *Describes a service with its service type, implementation, and lifetime*. In other words, this class guarantees that a class is treated as a service, with its lifetime and its type. The runtime takes charge of the list of this collection, and it will be protected.

From our point of view, we can shift the burden of safeguarding the object Singleton instance to dependency injection. This way, we avoid the antipattern behavior.

Entity Framework Core overview

As previously mentioned, this will just be a brief overview because we need to understand how to work with a database. This knowledge will be necessary for the design pattern explanation.

Entity Framework Core (EF Core) is an open-source, lightweight, extensible, and cross-platform version of Entity Framework, Microsoft's popular **Object-Relational Mapping (ORM)** framework, which was originally for .NET framework and now also for .NET9 as well. EF Core acts as a bridge between the database and C# code, allowing our application to manipulate database data using C# objects and LINQ queries without the need to write a lot of SQL code directly.

ORM

ORM facilitates interaction with a database using an object-oriented programming language. It enables developers to manage database operations through familiar object-oriented concepts instead of writing raw SQL statements, enhancing efficiency and reducing the likelihood of errors during the development process. ORMs improve code maintainability and simplify future updates or changes.

Features like caching, lazy loading, and connection pooling provided by ORMs can boost application performance. They create an abstraction layer that shields application code from changes in the underlying database schema, which aids in transitioning to different databases or scaling by distributing data across multiple servers. Overall, including ORMs in development frameworks enhances developer productivity, code maintainability, and application performance by allowing access to relational databases through recordset-like data structures.

Key features of Entity Framework Core

EF Core is equipped with several powerful features that make it a preferred choice for .NET developers working on data-driven applications. The following are some of its key features:

- **Tracking changes:** EF Core can track changes made to objects to issue the appropriate SQL statements when committing data back to the database.
- **LINQ support:** Allows querying using .NET's **Language Integrated Query (LINQ)**, making code easy to write and read.
- **Migrations:** Supports database migrations to manage changing database schemas by generating code.
- **Performance:** Optimized for performance and low memory consumption.

Using Entity Framework Core

Let us look at a simple project to understand the hands-on code of how we can use the EF Core with an in-memory database. The in-memory mode is a new way to interact with a database when the real database is not available. Remember that we are not going into unit testing. The in-memory database is not a good practice when making unit tests. Anyways, remember that it is possible to change the database provider to change the underlying database. First, we have to setup a project. In this case, as usual, we will use a Blazor Web APP project.

To use EF Core with an in-memory database, we must install one **nuget** package. From the Package Manager Console of Visual Studio

(View | Other Windows | Package Manager Console) run the following line:

Install-Package Microsoft.EntityFrameworkCore.InMemory

A simpler method is to go to Visual Studio Nuget Package manager for project and adding the **nuget**.

The data model

Now, we have to define the data model. A data model is a C# class or a set of classes that represent the database schema. For instance, if we have to manage our physical book library, we might have a **Book** class that represents a single book in our database and then on our bookshelf. This basic *domain class* has not yet been transformed into *entity* within the Entity Framework. In EF Core, *entities* correspond to specific tables in the database, and the framework monitors these entities. This tracking allows it to automatically carry out **Create, Read, Update, Delete (CRUD)** operations in the database, depending on the status of the objects. In our example, the **Book** class is as follows:

```
1. public class Book
2. {
3.     public int Id { get; set; }
4.     public string Title { get; set; }
5.     public string Author { get; set; }
6. }
```

Create the Database Context

To treat the **Book** class, which is a domain class, as an entity, it should be included as an EF Core **DbSet<T>** property within the **DbContext** class. **DbSet<T>** represents a collection of entities of a specific type, acting as a local representation of a database table. It allows for querying these entities using LINQ, tracks changes to these entities for CRUD operations, and synchronizes these changes with the database when **SaveChanges()** is called in the context. In short, it links the database tables to the entity classes in the application, facilitating data manipulation and synchronization.

In our class **DataContext**, we will represent the whole database. It derives from Entity Framework **DbContext**, which coordinates EF

functionality for a given data model, as we can see in the following code:

```
1. public class DataContext : DbContext
2. {
3.     public DbSet<Book> Books { get; set; }
4.     protected override void OnConfiguring
5.         (DbContextOptionsBuilder optionsBuilder)
6.         => optionsBuilder.UseInMemoryDatabase
7.             ("database");
8.     public void InitializeDatabase()
9.         => Database.EnsureCreated();
10. }
11.
```

Integrate the **DataContext** into the application's dependency injection, it is typically done in the **Program.cs**. Add the following line:

```
1. builder.Services.AddDbContext<DataContext>();
```

This ensures that whenever a **DataContext** is needed, the DI container provides an instance of our **DataContext** configured with the appropriate connection string (take references from [Chapter 6, Middleware and Extensibility](#) for details about DI).

Now that our database is connected and well configured as domain classes and entities, we can try to understand and craft a page or component that can interact with the above data. In our example, we will craft the home page in a Blazor Web App project, which can add a new book to the database and show the books stored in it.

The code is as follows:

```
1. @page "/"
2. @using Chapter8EFFactoryMethod.Models
3. @using Microsoft.EntityFrameworkCore
4. @inject DataRepository dbContext
5. @rendermode RenderMode.InteractiveServer
6. <PageTitle>Home</PageTitle>
7. <h1>My Books page</h1>
8. <style>
9.     .form-input {
```

```
10.     width: 100%;
11. }
12.
13. form {
14.     max-width: 300px;
15.     margin: auto;
16.     border: 2px solid #ccc;
17.     padding: 20px;
18.     box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
19. }
20. </style>
21. <h2> Add new book </h2>
22.     <EditForm      OnSubmit = "AddBook"      Model = "this"
    FormName = "myForm" >
23.     <label> Title </label>
24.         <input      class = "form-input"      type = "text"      @bind-
    value = "@NewTitle" /> <br />
25.     <label> Author </label>
26.         <input      class = "form-input"      type = "text"      @bind-
    value = "@NewAuthor" /> <br />
27.     <input class = "form-input" type = "submit" value = "Submit" /
    >
28. </EditForm>
29. <h3> Archived book list </h3>
30. @if (books == null)
31. {
32.     <p> <em> Loading... </em> </p>
33. }
34. else
35. {
36.     <ul>
37.         @foreach (var book in books)
38.         {
39.             <li> @book.Title by @book.Author </li>
40.         }
41.     </ul>
42. }
43.
```

```

44. @code {
45.     [SupplyParameterFromForm]
46.     private string NewTitle { get; set; } = string.Empty;
47.     [SupplyParameterFromForm]
48.     private string NewAuthor { get; set; } = string.Empty;
49.     private List<Book> books = new();
50.     protected override async Task OnInitializedAsync()
51.     {
52.         books = await dbContext.GetAllEntitiesAsync();
53.     }
54.     private async Task AddBook()
55.     {
56.         await dbContext.AddEntityAsync
57.             (new Book { Title = NewTitle, Author = NewAuthor });
58.         books = await dbContext.GetAllEntitiesAsync();
59.     }
60. }

```

The result will be as shown in [Figure 8.1](#):

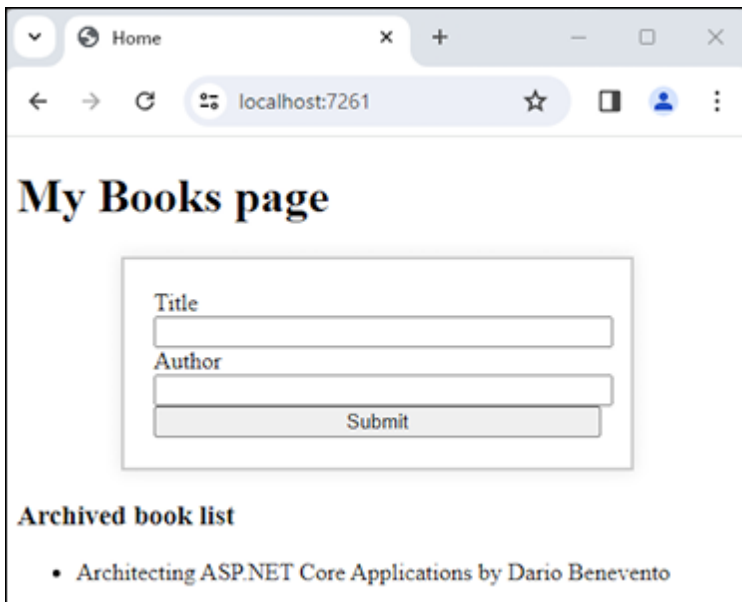


Figure 8.1: Sample book page

Following these steps, we can set up Entity Framework Core to

interact with a in-memory database, leveraging C# to handle all data operations. This setup is particularly useful for applications that require a lightweight database solution with the ease of an ORM.

Factory Method Pattern with EF Core

The Factory Method Pattern is a creational design pattern that defines an interface for creating objects but allows subclasses to alter the type of objects created. It provides a way to encapsulate object creation, enhancing code flexibility and reuse. In this pattern, a superclass specifies methods for creating objects, while subclasses implement these methods to create specific types of objects. This setup is beneficial when a class cannot anticipate the type of objects it needs to create or when object creation requires complex logic that should not be duplicated across classes.

Concepts

Using the Factory Method Pattern, we can adhere to the open-closed principle, where classes are open for extension but closed for modification. This pattern is commonly used in frameworks where library code needs to create objects of types defined by the client. It fosters maintainability and allows new object types to be added with minimal code changes.

How it works

The Factory Design Pattern is a creational pattern used to create objects without specifying the exact class of object that will be created. This is particularly useful in scenarios where the object setup is complex or when the configuration of objects needs to be flexible and easily changed. In the context of EF Core, the Factory Design Pattern can be utilized to manage the creation of data context objects, which is particularly useful in scenarios involving multiple databases or configurations.

Concrete problem

In our daily work as developers, we often encounter an annoying problem: debugging an application with a database connection

when we do not have physical access to the database. This situation can be frustrating as it prevents us from adequately testing and troubleshooting our code. In such scenarios, our progress can be hindered, leaving us with challenges such as limited access to data, encountering false or incomplete datasets, or having to spend precious time creating simulated environments. In such cases, we can figure out how to have a fake database to use.

Implementation

The aim of implementing the Factory Method Design Pattern is to use two different databases (physical database and in-memory), and the selection will be determined by a setting. So, the database will be set offline.

To achieve this objective, the following are the steps:

1. **Define the Factory interface:** This will declare the method(s) to create the **DbContext** object.
2. **Create concrete Factory classes:** These classes will implement the Factory interface to create specific **DbContext** instances.
3. **Configure dependency injection:** Set up the DI container to use the Factory for **DbContext** creation.
4. **Usage in application:** Use the Factory to obtain instances of **DbContext** in the application logic.

Before everything else, we have to install the part of EF Core that is able to connect with our database. To use EF Core with SQLite, we have to install three **nuget** packages as we have seen in previous paragraph dedicated to EF Core. Run the following lines:

Install-Package Microsoft.EntityFrameworkCore.Sqlite

Install-Package Microsoft.EntityFrameworkCore.InMemory

Install-Package Microsoft.EntityFrameworkCore.Design

These packages allow integration with SQLite and add design-time tools for migrations.

Define the Factory interface

The Factory interface for creating a **DbContext** is a design element

that abstracts the instantiation of **DbContext** instances, which is used for managing database operations in EF Core. This abstraction allows for centralized creation logic, flexibility in switching between different environment configurations, and improved testability through easy simulation of **DbContext** in unit tests. It also simplifies system maintenance and upgrades, promoting cleaner, more maintainable code. Our Factory will have a method to create the **DbContext** object and return the correct **DbContext** instance for the database type we have decided to use. To achieve this objective, the first step is to define an interface that will be common for the two classes and able to manage the two databases, as follows:

```
1. public interface IDbContextFactory
2. {
3.     MyDbContext CreateDbContext();
4.     void InitializeDatabase();
5. }
```

Create concrete classes

The concrete implementation will be made with two classes: the first for the SQLite database and the second for the in-memory database. Therefore, we can envision configuring a class for an SQL Server database or a PostgreSQL database in the same manner. The rest of the business logic will not change, provided that the database schema remains unchanged.

Create SQLite-specific Factory class

Now, let us implement the interface to provide a concrete Factory that creates **DbContext** instance configured for SQLite. The code is as follows:

```
1. public class SqliteDatabaseFactory : IDbContextFactory
2. {
3.     private readonly string connectionString;
4.     public SqliteDatabaseFactory(string connectionString)
5.     => this.connectionString = connectionString;
6.     public MyDbContext CreateDbContext()
7.     {
8.         var optionsBuilder = new
           DbContextOptionsBuilder<MyDbContext>();
```

```

9.     optionsBuilder.UseSqlite(connectionString);
10.    return new MyDbContext(optionsBuilder.Options);
11. }
12. public void InitializeDatabase()
13. {
14.     using var context = CreateDbContext();
15.     // Ensure the database is created
16.     context.Database.EnsureCreated();
17. }
18. }

```

The **SqliteDbContextFactory** class implements the **IDbContextFactory** interface to provide a method of creating instances of **MyDbContext** configured for an SQLite database. It initializes with a **connectionString** provided during construction, which it stores to construct a new **MyDbContext**. This class is created by **CreateDbContext** method, specifically setting up the SQLite database options. Additionally, there is an **InitializeDatabase** method that ensures the database is created if it does not already exist, leveraging the **CreateDbContext** method to handle the database setup and validation. This class encapsulates the details of database connection and creation, promoting the separation of concerns and making database operations easily manageable.

Create in-memory-specific Factory class

For the second type of database, we will use the in-memory database provided by EF Core. This option will be particularly useful when we debug our application without the actual database. However, it is important to note that while the in-memory database is valuable for development and debugging due to its simplicity and speed, it is not recommended for testing. Let us see how to implement this class, which provides a concrete Factory that creates **DbContext** instances configured for in-memory database. The code is as follows:

```

1. public class InMemoryDbContextFactory : IDbContextFactory
2. {
3.     private readonly string databaseName;
4.     public InMemoryDbContextFactory(string databaseName)
5.     => this.databaseName = databaseName;

```

```

6. public MyDbContext CreateDbContext()
7. {
8.     var optionsBuilder = new
DbContextOptionsBuilder< MyDbContext > ();
9.     optionsBuilder.UseInMemoryDatabase(databaseName);
10.    return new MyDbContext(optionsBuilder.Options);
11. }
12. public void InitializeDatabase()
13. {
14.     // For InMemory, initialization might
15.     // simply be ensuring the database can be accessed
16.     using var context = CreateDbContext();
17.     // Potentially seeding data or similar setup tasks
18. }
19. }

```

Like the previous class, we have to pass a **connectionString** to the Factory but in this case, it will be only the database name. The **InMemoryDbContextFactory** returns an appropriately configured **DbContext** for an in-memory database. In this case, ensuring the database initialization by accessing the **DbContext** is straightforward, we simply call the **CreateDbContext** method. The **InitializeDatabase** method is the ideal location to seed our in-memory database with initial or default data, preparing it for application debugging. If we do not seed it, we will encounter an empty database.

Configure dependency injection

Integrating the two Factory classes into the application's dependency injection setup is done in the **Program.cs** as usual. To distinguish which of the two Factory classes will be used, in this example, we will use simply a Boolean variable, but in a real case, it should be better to configure it in the **appsettings.json** so that we can configure the behavior of the application without touching the code, then without recompile. This part of our code will be as follows:

```

1. bool useInMemory = true;
2. if (useInMemory)
3. {

```

```

4.     builder.Services.AddScoped < IDbContextFactory > (provider
    =>
5.         new InMemoryDbContextFactory(databaseName));
6. }
7. else
8. {
9.     builder.Services.AddScoped < IDbContextFactory > (provider
    =>
10.        new SqliteDbContextFactory(connectionString));
11. }
12. builder.Services.AddScoped < DataRepository > ();

```

Typically, this part of the code should be placed just before the **builder.Build()** instruction. Furthermore, as we can see in the code, there are two defined variables: one for the connection string and the other for the in-memory database name. Those variables should arrive from the configuration and not be hardcoded as magic strings. Another point to add in the **Program.cs** file is the check for database creation. It is placed just before the **app.Run()**; and looks like the following snippet:

```

1. using (var scope = app.Services.CreateScope())
2. {
3.     var dbContext = scope.ServiceProvider
4.         .GetRequiredService < DataRepository > ();
5.     dbContext.InitializeDatabase();
6. }

```

Usage in application

In our application, the use of the Factory typically occurs at the repository or service layer, as follows:

```

1. public class DataRepository
2. {
3.     private readonly IDbContextFactory dbContextFactory;
4.     private readonly MyDbContext context;
5.     public DataRepository(IDbContextFactory dbFactory)
6.     {
7.         dbContextFactory = dbFactory;
8.         context = dbContextFactory.CreateDbContext();

```

```

9.     }
10.    public async Task<List<Book>> GetAllEntitiesAsync()
11.        => await context.Books.ToListAsync();
12.    public async Task AddEntityAsync(Book entity)
13.    {
14.        context.Books.Add(entity);
15.        await context.SaveChangesAsync();
16.    }
17.    public void InitializeDatabase()
18.        => context.Database.EnsureCreated();
19. }

```

In this example, **DataRepository** uses the Factory to create a **DbContext** instance. This approach is specifically useful for managing database operations across different parts of the application, ensuring that the database context is correctly configured for the desired database engine. The repository pattern and how to use it properly has been covered in [Chapter 3, Architectures and Core Components](#), in a well-focused paragraph.

Using the Factory Design Pattern with EF Core, we gain a flexible way to manage database contexts and their configurations. This approach is particularly useful in large applications or systems where different parts of the application might require different database configurations or types or to avoid installing a database on the development machine.

Decorator Pattern for logging

The Decorator Pattern primarily aims to dynamically add new responsibilities to objects without relying on subclassing.

Concepts

This pattern offers several key advantages. Firstly, it provides enhanced flexibility, allowing for functionalities to be added or modified at runtime, which differs from the rigidity of static inheritance. Additionally, it helps prevent class proliferation in complex systems by enabling functionalities to be incrementally added, thus avoiding a multitude of subclasses. Furthermore, the Decorator Pattern aligns with the Open/Closed Principle, one of the

important SOLID principles, as it allows objects to be extended without modifying their existing structure.

How it works

The Decorator Pattern involves a set of component classes and decorators that improve these components. At its heart, there are a few essential parts. The component, an interface or an abstract class, sets the rules for objects to take on new roles as needed. A concrete component follows this set of rules and provides the basic features. A Decorator keeps a reference to a component object and follows its rules. The Decorator adds its behavior either before or after passing the task to the component. Concrete decorators are specific types of decorators that change or improve the component's behavior in particular ways. Let us expand our previous example with more details, showing how to implement and utilize the Decorator Pattern for a logging feature in a C# application.

Concrete problem

During the maintenance of existing software, or in the phase known as *software gardening*, we often face challenges that require us to add functionality to a class, potentially leading to instability. This situation is common when software evolves, or minor adjustments are needed to improve performance or add features. In the example we will tackle in this section, we need to implement logging in an existing class without altering the original class. This requirement often arises when you want to track or monitor specific behaviors for debugging or analytics purposes. Making direct changes to the original class can be risky, especially if the class is widely used or if any changes could inadvertently affect other parts of the application. To address this issue elegantly, the Decorator Pattern comes to our aid. This pattern allows us to dynamically enhance the behavior of objects without affecting other components or violating the principles of good design. Let us delve into how the Decorator Pattern can help us achieve this.

Implementation

In software design, particularly within the Decorator Pattern context, the concept of a basic interface serves as a contract that

defines a set of operations or behaviors that a service must provide. This foundational interface allows developers to create services that adhere to a specific blueprint, ensuring consistency across implementations. The Decorator Pattern leverages this concept to add new behaviors dynamically without altering the original service's structure.

The interface our original object depends on, could be as follows:

```
1. public interface IService
2. {
3.     string PerformOperation();
4. }
```

The implementation of this interface in the original object is straightforward. **IService** defines a simple operation, as follows:

```
1. public class ConcreteService : IService
2. {
3.     public string PerformOperation()
4.         => "Operation Performed";
5. }
```

We will return a string in this case, but we can imagine any other return or process.

Dynamic extension of operations offers a robust way to enhance a service's behavior without altering its original code. The Decorator Pattern facilitates this by wrapping additional functionalities around an existing service. For example, to log the execution of a method such as **PerformOperation**, an abstract class can serve as a Decorator, implementing **IService** while accepting an **IService** instance as a dependency. This allows for runtime behavior modifications, keeping the core service intact while augmenting it with new capabilities. The resulting architecture is modular, flexible, and easy to maintain, adapting dynamically to changing requirements or business needs. The following is the code for our abstract service Decorator:

```
1. public abstract class ServiceDecorator : IService
2. {
3.     protected IService service;
4. }
```

```

5. public ServiceDecorator(IService service)
6. {
7.     this.service = service;
8. }
9. public virtual string PerformOperation()
10. {
11.     return service.PerformOperation();
12. }
13. }

```

The concrete logging Decorator will enhance the **IService** by adding log messages before and after an operation. The Decorator and service share an interface, with the Decorator wrapping around the service to maintain core behavior while adding logging functionality. Our concrete Decorator will be as follows:

```

1. public class LoggingDecorator : ServiceDecorator
2. {
3.     public LoggingDecorator(IService service) : base(service) { }
4.     public override string PerformOperation()
5.     {
6.         Log("Operation started.");
7.         var result = base.PerformOperation();
8.         Log("Operation ended.");
9.         return result;
10.    }
11.    private void Log(string message)
12.    {
13.        Console.WriteLine($"Log: {message}");
14.    }
15. }

```

To ensure our Decorator class will be available for the whole application, we have to add it to DI and in particular, in the **Program.cs**, where we have to build a **LoggingDecorator**. The code is as follows:

```

1. builder.Services.AddScoped<IService>(provider =>
2. {
3.     var service = new ConcreteService();
4.     return new LoggingDecorator(service);

```



```
5. });
```

Services can be decorated as they are registered in the application's service container, ensuring that any component or service requesting these services will receive the decorated version. We will inject the decorated service into our Blazor components to use the enhanced functionality. In the following code, we will see it in *line 2*:

```
1. @rendermode InteractiveServer
2. @inject IService LoggedService
3.
4. <h1>Decorator Pattern for logging</h1>
5.   <button @onclick="PerformLoggedOperation">Perform
   Operation</button>
6. <p> @message </p>
7.
8. @code {
9.     private string message = string.Empty;
10.    private void PerformLoggedOperation()
11.    {
12.        message = LoggedService.PerformOperation();
13.    }
14. }
```

This approach centralizes the logging logic and keeps the rest of the application clean and focused on its primary responsibilities, demonstrating the power of the Decorator Pattern in maintaining clean and maintainable code architectures.

Observer Pattern for application state

The Observer Pattern is a behavioral design pattern that establishes a one-to-many dependency between objects, allowing multiple observers to listen to and react to changes in a single subject. In this pattern, a subject maintains a list of its dependent observers, notifying them of any state changes.

Concepts

This pattern is useful when an object's state change needs to be communicated to multiple other objects without tightly coupling

them. The Observer Pattern promotes loose coupling and separation of concerns, as the subject and observers can evolve independently.

How it works

A classic use case of the Observer Pattern is in user interfaces, where multiple components such as buttons, labels, and forms need to update simultaneously in response to changes in data or user inputs. In this model, a central data source, or *subject*, monitors for any changes. When a change occurs, it automatically notifies all registered components, known as *observers*. These observers are responsible for updating their appearance or state to reflect the new information, ensuring the user interface remains consistent and interactive.

Concrete problem

One of the most difficult and tedious issues regarding a web application is managing cross-sectional state across all pages. If we consider the front end, we have several ways built-in to the framework for managing the state within our components, regardless of the hosting model chosen. The methods are as follows:

- Cascading values and event callbacks
- State container with Observer Pattern

To achieve this goal, we will utilize the Blazor Web App project template. It is crucial to correctly set the *Interactive render mode* parameter during the project creation phase by choosing from the following:

- None
- Server
- WebAssembly
- Auto (Server and WebAssembly)

The choice of this parameter will change our approach to some solutions. In our case, by selecting **Server** (then **server-side rendering (SSR)**), the transmission mode of the **CascadingParameters** varies. The **CascadingParameters** do not pass data across render mode boundaries. Interactive sessions operate in a different context compared to pages that utilize static SSR. There

is no guarantee that the server rendering the page is the same one that will host a subsequent Interactive Server session, especially in scenarios involving WebAssembly components where the server and client are on separate machines. The advantage of static SSR is that it leverages the total efficiency of stateless HTML rendering.

Data that transition from static to interactive rendering modes must be capable of serialization. Components, which are complex objects linked to an extensive network of other entities, including the renderer, the DI container, and every instance of a DI service, require explicit serialization from static SSR to be accessible in interactively rendered components.

There are two main methods used, as follows:

- Within the Blazor framework, parameters crossing from static SSR to interactive render modes are automatically serialized if they are JSON-serializable; otherwise, an error occurs.
- State preserved in **PersistentComponentState** is also serialized and restored automatically if it is JSON-serializable; if not, an error occurs.

CascadingParameters are not JSON-serializable because they typically function similarly to DI services. Often, cascading parameters have platform-specific variations, and forcing developers to standardize these across server and client interactions or within WebAssembly could hinder development. Additionally, many cascading parameter values are inherently non-serializable, making it impractical to mandate changes in existing applications that use nonserializable cascading parameters.

To ensure the state is accessible across all interactive components via cascading parameters, it is advisable to use root-level cascading values. A Factory Pattern can be implemented to generate and dispatch updated values after the application launch. These root-level cascading values are then accessible to all components, including those in interactive modes, as they are treated like DI services. The following figure will show how **CascadingParameter** and **EventCallback** work together in an application:

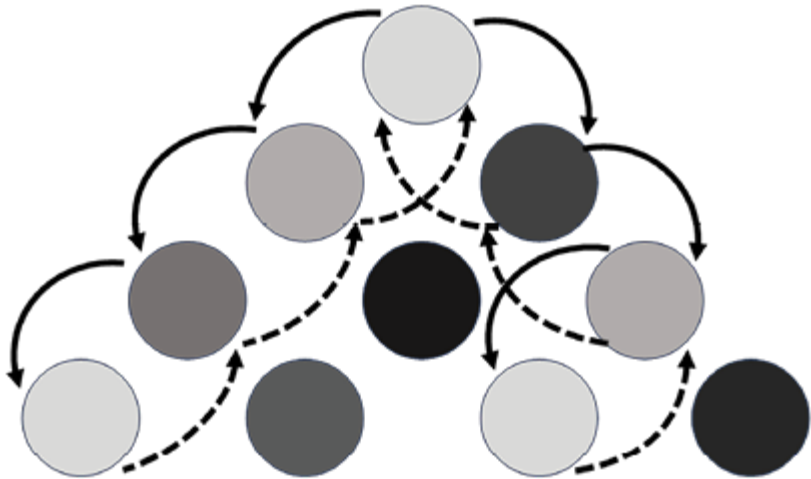


Figure 8.2: Cascading values and event callbacks

The black arrows descended in full suckling represent the **CascadingValues**, while the dotted arrows represent the events. Two things to be noted are as follows:

- It is very easy to create confusion.
- The components that have nothing to do with the parameter are obliged to manage it.

Managing the state across various backend modules and notifying different parts of the system about state changes is a crucial design consideration. For modules that are separate and potentially distributed, using messages over sockets is a viable approach. This method facilitates asynchronous communication and efficiently handles state updates across different services or applications that are not directly connected.

However, direct method calls and events are commonly employed for interacting nested objects within the same application domain. Although these approaches are straightforward and allow for real-time updates, they also introduce tight coupling between the modules and objects involved. Such tight coupling can make the system less flexible and harder to maintain, as changes in one part of the system might require corresponding modifications in all connected components.

Implementing the Observer Pattern can significantly enhance scalability and maintainability in the backend scenarios or even in

the front end when using ASP.NET Core. This pattern reduces dependencies between the entities maintaining the state and the observers' needing notifications about changes, thereby promoting more loosely coupled components. This structural approach simplifies the overall system design and improves adaptability to changing requirements or technologies, making it a robust choice for complex software architectures.

Implementation

Consider a frontend scenario where our Blazor application or ASP.NET Core application supports the dark and light theme. This parameter should ideally be associated with the user, and we would extend the theme across the pages. The method we chose for this example to solve this problem is the Observer design pattern. For the first step, we will craft a class that stores the theme as follows:

```
1. public class ThemeManager
2. {
3.     private string currentTheme;
4.     public string CurrentTheme
5.     {
6.         get { return currentTheme; }
7.         set
8.         {
9.             if (currentTheme != value)
10.            {
11.                currentTheme = value;
12.                NotifyObservers();
13.            }
14.        }
15.    }
16. }
```

Another fundamental class will be the **ObserverManager**. This class is intended to register and deregister all clients and notify them of any changes. The implementation of the example taken will be as follows:

```
1. public class ObserverManager
2. {
3.     private List<IObserver> observers = new
```

```

List<IObserver> ();
4.
5. public void RegisterObserver(IObserver observer)
6. {
7.     observers.Add(observer);
8. }
9.
10. public void UnregisterObserver(IObserver observer)
11. {
12.     observers.Remove(observer);
13. }
14.
15. public void NotifyThemeChanged(string theme)
16. {
17.     foreach (var observer in observers)
18.     {
19.         observer.Update(theme);
20.     }
21. }
22. }

```

Now that we have the class for theme manager (the observable), we have to create the observers' classes. We have our Blazor Web App project, and we implement the above class. We will instantiate it as Singleton as we have seen before for convenience.

In this case, we have an interface that describes the observable class, and our ASP.NET Core dependency injection accepts it as a generalization. The code is as follows:

```

1. var builder = WebAssemblyHostBuilder.CreateDefault(args);
2.
3. builder.Services.AddSingleton<ThemeManager>();
4. builder.Services.AddSingleton<ObserverManager>();
5.
6. await builder.Build().RunAsync();
7. // etc...

```

After registering the two singletons, all the pages should have an injection of the two classes. In our Blazor project, we will do that in the razor page or component as follows:

1. @inject ThemeManager ThemeManager

In this case, the engine of Blazor will inform all the components that a property of an injected object has changed. If we have another object (for example, a class) that needs to be informed that something has changed, it should inherit from **IObserver**, as follows:

```
1. public interface IObserver
2. {
3.     string Update(string theme);
4. }
```

The Observer class should be as follows:

```
1. public class MyThemeObserver : IObserver, IDisposable
2. {
3.     ThemeManager ThemeManager;
4.     ObserverManager ObserverManager;
5.     bool disposed;
6.
7.     public string MyClassTheme { get; set; }
8.
9.     public MyThemeObserver(ThemeManager themeManager,
ObserverManager observerManager)
10.    {
11.        ThemeManager = themeManager;
12.        ObserverManager = observerManager;
13.
14.        ObserverManager.RegisterObserver(this);
15.    }
16.
17.    public string Update(string theme) => MyClassTheme =
theme;
18.
19.    public void Dispose()
20.    {
21.        Dispose(true);
22.        GC.SuppressFinalize(this);
23.    }
24.
25.    protected virtual void Dispose(bool disposing)
```

```
26. {
27.     if (disposed) return;
28.
29.     if (disposing)
30.         ObserverManager.UnregisterObserver(this);
31.
32.     disposed = true;
33. }
34. }
35.
```

The **Observer** class should implement **IDisposable** properly to unregister from **ObserverManager**. The correct way to implement the **IDisposable** pattern is shown before. The primary purpose behind implementing the **Dispose** method is to free unmanaged resources. When dealing with instance members that implement **IDisposable**, it is typical to chain **Dispose** calls. Additionally, there are various other motives for implementing **Dispose**, such as releasing allocated memory, removing items from a collection, or indicating the release of an acquired lock. In our case, the purpose of the **dispose** method is to release the class from **ObserverManager**'s subscription.

Furthermore, in this example, we can find an additional **Dispose** method marked as *protected virtual* because we implement the *dispose pattern*. Within the overload, the **disposing** parameter serves as a Boolean indicator, distinguishing whether the method invocation originates from a **Dispose** method (denoted by a true value) or a finalizer (denoted by a false value). Any non-sealed classes should be regarded as potential base classes since they may be inherited. When implementing the **dispose** pattern for any such potential base class, the following must be provided:

- A **Dispose** is implementation that calls the **Dispose(bool)** method.
- A **Dispose(bool)** method is tasked with carrying out the necessary cleanup operations.

Properly disposing of all classes is a good practice and a must be followed by every programmer to avoid unexpected behaviors and memory leaks.

In the Observer Pattern, it is crucial to ensure that the **Observer** class is properly disposed of.

Strategy Pattern for managing payments

The Strategy Pattern is a behavioral design pattern that allows the selection of an algorithm at runtime. This pattern defines a family of algorithms, encapsulates each as a Strategy, and makes them interchangeable. The key is to define a common Strategy interface for all algorithms and allow different algorithms to be plugged into a context. The context, which is the main class, maintains a reference to the Strategy and delegates the algorithm's execution to it.

Concepts

This pattern promotes flexibility and maintainability by enabling the selection of algorithms dynamically, which helps avoid complex conditional logic. It is commonly used when a class has multiple algorithms for a specific behavior, such as sorting or formatting. The Strategy Pattern adheres to the open-closed principle, allowing the system to be extended with new strategies without modifying existing code, making it ideal for applications requiring adaptable or variable behaviors.

How it works

The Strategy Pattern works by defining a common interface for a family of algorithms, which are encapsulated as individual classes implementing this interface. The pattern involves three main components: the Strategy interface, concrete strategies, and the context class.

- **Strategy interface:** This defines the method(s) each concrete Strategy will implement. It is the contract that ensures consistency among different algorithms.
- **Concrete strategies:** These classes implement the Strategy interface, each providing a different algorithm. They encapsulate the specific behaviors that can be selected at runtime.
- **Context class:** The context class maintains a reference to a

Strategy and delegates the algorithm's execution to it. The client code can dynamically switch the Strategy used by the context.

By separating algorithms into distinct classes, the Strategy Pattern allows for flexibility and clean code, avoiding complex conditional logic and enabling the addition of new strategies without altering existing ones.

Concrete problem

In an e-commerce business, managing multiple payment methods is crucial for accommodating diverse customer preferences. Integrating various options like credit cards, PayPal, and digital wallets often results in complex code due to distinct APIs, validation requirements, and security measures for each type. As the number of payment methods grows, so does the complexity, leading to code that is hard to maintain and error-prone. This issue is exacerbated when conditional logic is used to manage different payment types, creating a tightly coupled system. Modifying or adding new payment methods usually requires significant code changes, affecting scalability and adaptability.

Implementation

Implementing a Strategy Pattern for managing payment strategies in a C# application with Blazor must follow a few key steps. We will create a base interface for payment strategies, implement several concrete strategies, and then use these strategies in a specific context, such as a Blazor page.

1. First, let us define an **IPaymentStrategy** interface that will have a method to process payments. The code is as follows:

```
1. public interface IPaymentStrategy
2. {
3.     Task ProcessPaymentAsync(decimal amount);
4. }
5.
```

2. Now, create concrete strategies. In this example, we have to implement two strategies: **CreditCardPayment** and **PaypalPayment**, but we can create all the strategies we

want. The most important thing in this context is that each Strategy class must implement **IPaymentStrategy** as follows:

```
1. public class CreditCardPayment : IPaymentStrategy
2. {
3.     public async Task ProcessPaymentAsync(decimal
amount)
4.     {
5.         // Logic for processing credit card payment
6.         Console.WriteLine($"Processing Credit Card
7.             payment for {amount:C}");
8.     }
9. }
10. public class PaypalPayment : IPaymentStrategy
11. {
12.     public async Task ProcessPaymentAsync(decimal
amount)
13.     {
14.         // Logic for processing PayPal payment
15.         Console.WriteLine($"Processing PayPal
16.             payment for {amount:C}");
17.     }
18. }
```

3. The payment context will keep track of the current Strategy and allow changing it dynamically, as follows:

```
1. public class PaymentContext
2. {
3.     private IPaymentStrategy paymentStrategy;
4.     public PaymentContext(IPaymentStrategy
paymentStrategy)
5.     {
6.         this.paymentStrategy = paymentStrategy;
7.     }
8.     public void SetPaymentStrategy(IPaymentStrategy
paymentStrategy)
9.     {
```

```

10.     this.paymentStrategy = paymentStrategy;
11. }
12. public async Task ExecutePayment(decimal amount)
13. {
14.     await
        paymentStrategy.ProcessPaymentAsync(amount);
15. }
16. }

```

4. The two strategies and the context must be registered in our dependency injection as follows:

```

1. builder.Services.AddSingleton<PaymentContext>();
2. builder.Services.AddSingleton<IPaymentStrategy,
    PayPalPayment>();
3. builder.Services.AddSingleton<IPaymentStrategy,
    CreditCardPayment>();

```

5. Take note of the registration of the two strategies; we have one interface and two classes registered. To retrieve the right strategy in our Blazor component, we have to inject the collection of our interface as follows:

```

1.     @inject      IEnumerable<IPaymentStrategy>
        PaymentStrategies

```

6. Then, retrieve the correct instance as follows:

```

1.     PaymentStrategies.FirstOrDefault(p => p is
        CreditCardPayment);

```

In the ASP.NET Core dependency injection, when we register more than one class on a single interface and try to retrieve it as usual with **@inject IPaymentStrategy**, the returned class will be the last registered. By injecting the **IEnumerable<IPaymentStrategy>** we will have the whole list of registered classes on our interface.

7. Next, we have to implement a page that allows us to view the chosen payment. In our Blazor page, we can use the **PaymentContext** to process payments with the selected Strategy from the user, as follows:

```

1. @page "/"

```

```

2. @using Chapter8Strategy.Models
3.     @inject      IEnumerable < IPaymentStrategy >
      PaymentStrategies
4. @rendermode InteractiveServer
5. <PageTitle> Home </PageTitle>
6. <h1> Chapter 8 - Strategy Pattern </h1>
7. <h3> Choose Payment Method </h3>
8. <select @onchange = "ChangePaymentStrategy" >
9.     <option value = "CreditCard" > Credit Card </option>
10.    <option value = "PayPal" > PayPal </option>
11. </select>
12.    <button      @onclick = "()"           = >
      ProcessPayment(100M)" > Pay $100 </button>
13. @code {
14.     private PaymentContext paymentContext;
15.     protected override void OnInitialized()
16.     {
17.         // Set a default strategy
18.         paymentContext = new PaymentContext
19.             (PaymentStrategies.FirstOrDefault
20.              (p => p is CreditCardPayment));
21.     }
22.     private void ChangePaymentStrategy(ChangeEventArgs
23.     e)
24.     {
25.         if (e.Value.ToString() == "CreditCard")
26.             paymentContext.SetPaymentStrategy
27.                 (PaymentStrategies.FirstOrDefault
28.                  (p => p is CreditCardPayment));
29.         else
30.             paymentContext.SetPaymentStrategy
31.                 (PaymentStrategies.FirstOrDefault
32.                  (p => p is PaypalPayment));
33.     }
34.     private async Task ProcessPayment(decimal amount)
      = >      await

```

```
paymentContext.ExecutePayment(amount);  
35. }
```

With this setup, the Blazor application can easily switch payment methods on the fly, demonstrating how the Strategy Pattern can be helpful in managing various behaviors in a flexible and maintainable way.

Conclusion

In this chapter, we have explored the practical application of GoF design patterns in ASP.NET Core, focusing on the key patterns and their usage in modern web development. The Singleton Pattern emerged as a valuable tool for managing shared resources in ASP.NET Core applications, ensuring a single instance of critical services. The Factory Method Pattern demonstrated its efficacy in EF Core, enabling dynamic creation of object instances and fostering flexibility in data access layers. The Decorator Pattern showcased its utility in enhancing functionality, particularly for logging, without modifying existing code. The Observer Pattern proved effective for handling application state changes, allowing multiple components to react to updates seamlessly. Finally, the Strategy Pattern highlighted its importance in managing payments, offering a clean and flexible approach to switching payments algorithms.

By leveraging these patterns, we can create scalable, maintainable, and robust ASP.NET Core applications, demonstrating the continued relevance of classic design patterns in addressing modern software challenges. In this chapter, we focused on the front end simply because the effects are most visible and the comprehension is immediate.

Points to remember

- **Singleton Pattern:** Ensures a class only has one instance and provides global access to it.
- **Factory Method Pattern:** It allows a class to defer object creation to its subclasses.
- **Decorator Pattern:** It adds new functionalities to objects

dynamically, enhancing them without modifying their structure.

- **Observer Pattern:** It enables a subject to notify all its observers about changes, maintaining a list of dependents.
- **Strategy Pattern:** It facilitates the choice of an algorithm at runtime from a predefined set of algorithms.

Multiple choice questions

1. **What is the primary purpose of using GoF design patterns in ASP.NET Core?**
 - a. To improve the performance of applications
 - b. To increase the security of applications
 - c. To enhance the maintainability and scalability of applications
 - d. To reduce the storage requirements of applications
2. **Which design pattern ensures that a class has only one instance and provides a global point of access to it in ASP.NET Core?**
 - a. Factory Method Pattern
 - b. Singleton Pattern
 - c. Observer Pattern
 - d. Strategy Pattern
3. **What does the Factory Method Pattern primarily solve in the context of EF Core?**
 - a. Managing database connections
 - b. Creating objects without specifying the exact class of object that will be created
 - c. Logging queries to a database
 - d. Encrypting data before saving it to the database
4. **Which design pattern allows objects to be modified dynamically by wrapping them with more features?**

- a. Strategy Pattern
- b. Singleton Pattern
- c. Decorator Pattern
- d. Observer Pattern

5. Which aspect of an application is the Observer Pattern useful for?

- a. Handling user authentication
- b. Managing state changes throughout the application
- c. Creating single instances of a class
- d. Connecting to different types of databases

6. Which pattern is characterized by creating a hierarchy of classes which are related by a series of derived classes?

- a. Decorator Pattern
- b. Singleton Pattern
- c. Factory Method Pattern
- d. Observer Pattern

7. How does the Singleton Pattern prevent the creation of additional instances?

- a. By using a public constructor
- b. By making the class constructor private and providing a static method to get the instance
- c. By using multiple constructors with different parameters
- d. By locking the object during its creation

8. What type of design pattern is the Decorator Pattern classified as?

- a. Behavioral
- b. Structural
- c. Creational
- d. Functional

9. Which pattern involves subscribers that are notified of

changes in the object they are observing?

- a. Singleton Pattern
- b. Factory Method Pattern
- c. Decorator Pattern
- d. Observer Pattern

Answer key

Key terms

- **GoF:** *Gang of Four* is the name given to *Erich Gamma, Helm Richard, Johnson Ralph* and *Vlissides John*, the authors of *Design Patterns: Elements of Reusable Object-Oriented Software*.
- **Design pattern:** A reusable solution to a common problem encountered in daily software development.
- **Creational design patterns:** Creational design patterns offer different mechanisms for creating objects, enhancing flexibility, and promoting the reuse of existing code.
- **Structural design patterns:** Structural design patterns describe methods for organizing objects and classes into broader configurations that maintain their flexibility and efficiency.
- **Behavioral design patterns:** Behavioral design patterns focus on organizing algorithms and distributing responsibilities among objects.

1 Design Patterns - Elements of Reusable Object-Oriented Software (Chapter 1.1 from Christopher Alexander)

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 9

CQRS in Architecture

Introduction

In this chapter, we will explore the **Command Query Responsibility Segregation (CQRS)** architectural pattern, a Strategy designed to enhance efficiency and scalability in complex software systems. By separating read and write operations into distinct models, CQRS allows for optimization of each operation, improving system performance and simplifying maintenance. But this makes sense only for big systems. Before choosing this pattern, we have to be absolutely careful and evaluate its complexity in the application we are building.

For the first time, we will examine CQRS and then **event sourcing**. After this first excursion, we will put together the two concepts, making what in architecture is called **CQRS with event sourcing (CQRS-ES)**. This is an approach that records changes as a series of events, enabling robust, history-based operations. Each status of each aggregate will be rebuilt from all events in his history. This combination is critical for creating detailed audit trails and supporting complex business transactions.

The final section of this chapter covers the practical implementation of CQRS. We will explore two examples of real-world application

setups, thoroughly addressing the common challenges that developers encounter and outlining effective strategies for maintaining data consistency across distributed systems.

Structure

In this chapter, we will cover the following topics:

- Introduction to CQRS
- Command and query separation
- Event sourcing integration
- Implementing CQRS in practice

Objectives

The primary objectives of this chapter are to provide a quick overview of the CQRS pattern and to demonstrate its practical applications. We aim to introduce the core principles of CQRS and the advantages of separating command and query functions in software architectures. The chapter will also explore the integration of CQRS-ES, highlighting how this combination enhances system reliability and traceability. Furthermore, we will offer practical guidance on implementing CQRS-ES in sample projects, addressing common challenges and strategies for maintaining data consistency. Through theoretical concepts and real-world examples, this chapter helps facilitate the adoption of CQRS in efficient and scalable systems.

Introduction to CQRS

CQRS is an architectural pattern used in software design to separate the process of writing data (**commands**) from the process of reading data (**queries**). This approach is based on the principle of separating concerns: instead of using a single model for both writing and reading data operations, it advocates for having one model to handle commands (that is, updating the application state) and another model to handle queries (that is, querying and viewing the status of the application) as we can see in the following figure:

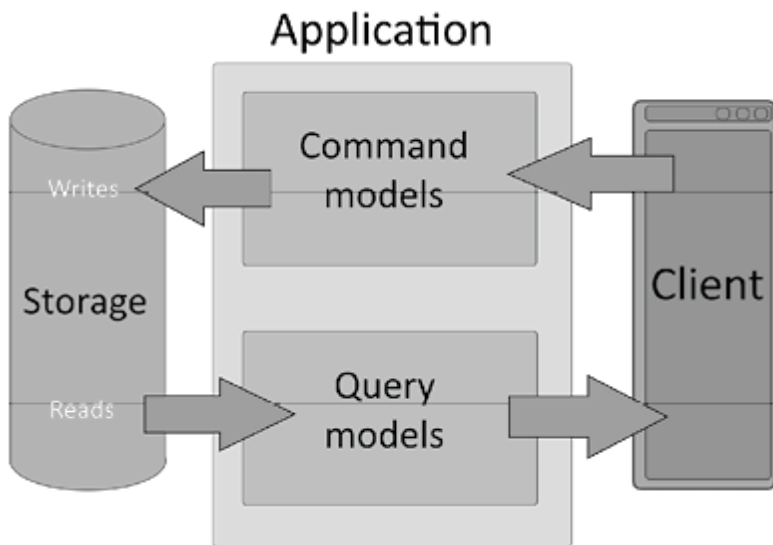


Figure 9.1: CQRS basic block diagram

In practice, this often translates into using two distinct models: one to manage the application state in response to commands, known as the **domain model** or **write model**, and another for read-only data access, known as the **query model** or **read model**. These two models can be optimized for their respective responsibilities, allowing for greater flexibility and scalability in system implementation.

CQRS also promotes the use of physical separation between the write and read systems, enabling resource optimization and horizontal scalability. This can lead to better load management, as it allows for independently scaling the write and read systems based on application needs.

Furthermore, CQRS can facilitate the design of complex systems by allowing for separate management of business logic and data querying. However, it is important to note that implementing CQRS introduces additional complexity to the system and requires careful planning and design to ensure significant benefits.

Command and query separation

As we said in the introduction, in CQRS, a key concept is the clear

distinction between two information flows: commands, which move from client to server, and queries, which flow from server to client. This separation is essential to its architecture, categorizing operations into these two distinct types for clarity and efficiency, as follows:

- **Commands:** It represents actions that intend to change the state of the system. They are responsible for initiating updates to the application state, such as creating, updating, or deleting data. Commands encapsulate the intention to perform an action, often containing all the necessary information to execute that action.
- **Queries:** They are operations that retrieve data from the system without altering its state. They are responsible for reading and presenting data to users or other parts of the system. Queries do not cause any side effects or modifications to the application state; they solely provide information.

This clear distinction between commands and queries enables software architects and developers to design and implement specialized components optimized for their respective tasks. For example, the **write** side of the system can focus on enforcing business rules, maintaining consistency, and handling concurrency. In contrast, the **read** side can be optimized for efficient data retrieval and query processing.

Overall, command and query separation within CQRS promotes a cleaner and more maintainable architecture by clearly delineating between operations that modify state and operations that retrieve data, leading to better scalability, performance, and flexibility in system design.

Event sourcing integration

Event sourcing is an architectural pattern that often works with the CQRS approach by capturing all changes to an application state as a sequence of events. This method involves storing these events as the primary source of truth rather than the current state, which provides a comprehensive historical record. In event sourcing, events are stored in a specialized event store, a database designed to

handle append-only data storage.

One of the fundamental benefits of event sourcing is its ability to provide a complete audit trail. Since every change is recorded as an event, it offers an accurate and exhaustive log of all actions taken within the system, which is essential for compliance and regulatory needs. Moreover, the system's capability to replay events lends itself well to debugging and correcting errors in the state without losing data integrity.

However, implementing event sourcing introduces added complexity to the development process. We must manage event versioning and ensure consistent application of events across the system. Another challenge is event migration; as the system evolves, the structure of the events might need updates, which can be a complex process if not handled carefully. Additionally, the performance implications of replaying a large number of events to restore the state must be considered. This often leads to implementing strategies like snapshots of the state at intervals to reduce the number of events that need replaying.

Implementing CQRS in practice

In this section, we will explore two use cases: the first one will be an example of how to use the CQRS pattern in an e-commerce that manages product information; the second will be an expansion of the first example with the integration of the event sourcing concept in the e-commerce application of the first example. There will be a third example that shows data from the two previous applications with CQRS-ES with Blazor as the front end.

Product management

Let us go through a simple implementation example of CQRS using C# and ASP.NET Core, focusing on a web application that manages product information. We will separate read operations (queries) from write operations (commands) using distinct models for each.

Project structure

To achieve this goal, we have to build a new solution in Visual Studio 2022 with some logical divisions inside. The project will

contain the following:

- **Web API:** ASP.NET Core Web API project to handle HTTP requests.
- **Domain models:** Core domain entities and business logic.
- **Application:** Contains business logic and CQRS models.
- **Infrastructure:** Concrete data access implementation.

Now we can analyze in depth each project.

Web API

Create a new ASP.NET Core Web API project in Visual Studio or using the CLI command:

```
dotnet new webapi -n Chapter9CQRS-API
```

or simply create the project with Visual Studio wizard. Now, we have to add some dependencies to better manage CQRS. The most used package for CQRS implementation is **MediatR**. This is a .NET library that implements the mediator pattern, facilitating object interactions by routing commands, queries, and events to corresponding handlers, simplifying decoupling in applications and improving separation of concerns and code organization. For more details on **MediatR**, please refer to their official website.

Install necessary packages with the following scripts:

```
dotnet add package MediatR
```

```
dotnet add package Swashbuckle.AspNetCore
```

even if in .NET9 Swagger is not added by default, we can use it. In this case our **Program.cs** will be as follows:

```
1. var builder = WebApplication.CreateBuilder(args);
2. builder.Services.AddEndpointsApiExplorer();
3. builder.Services.AddSwaggerGen();
4. builder.Services.AddControllers();
5. builder.Services.AddMediatR(cfg => cfg
6.     .RegisterServicesFromAssembly(
7.         typeof(CreateProductHandler).Assembly));
8. builder.Services.AddMediatR(cfg => cfg
9.     .RegisterServicesFromAssembly(
10.        typeof(GetProductByIdHandler).Assembly));
```



```

11. var app = builder.Build();
12. if (app.Environment.IsDevelopment())
13. {
14.     app.UseSwagger();
15.     app.UseSwaggerUI();
16. }
17. app.UseHttpsRedirection();
18. app.Run();

```

Domain models

To effectively handle a product, a model is necessary for its management. To achieve this, in our project add a **Models** folder and define the following simple class:

```

1. public class Product
2. {
3.     public int Id { get; set; }
4.     public string? Name { get; set; }
5.     public decimal Price { get; set; }
6. }

```

Application

At this point, we must divide the two paths. In a common folder called **Features**, we create the two paths through two other folders: a Commands folder and a Queries folder. Here, we will place our classes for commands and queries. We will, therefore, define the command as follows:

```

1. using MediatR;
2. namespace Chapter9CQRS_API.Features.Commands;
3. public class CreateProductCommand : IRequest<int>
4. {
5.     public string Name { get; set; }
6.     public decimal Price { get; set; }
7. }

```

Then, we will implement a simple query as follows:

```

1. using Chapter9CQRS_API.Models;
2. using MediatR;
3.

```

```

4. namespace Chapter9CQRS_API.Features.Queries;
5.
6. public class GetProductByIdQuery : IRequest<Product>
7. {
8.     public int Id { get; set; }
9. }

```

At this point, the two managers for the two paths are ready.

Infrastructures

Since we have chosen MediatR as our library, we need to utilize two specific classes known as **Handlers** to manage commands and queries.

- **Implement Command Handlers:** Handlers are essential for processing the data flow within our application. In the following code, we will outline a simple approach to implementing these Handlers, but without logic to store data. It is only an example of the command handler:

```

1. public class CreateProductHandler
2.     : IRequestHandler<CreateProductCommand, int>
3. {
4.     public Task<int> Handle(
5.         CreateProductCommand request
6.         , CancellationToken cancellationToken)
7.     {
8.         // Here you would add your logic
9.         // to save the product to the database
10.        // For simplicity, let's assume
11.        // we're returning a random ID
12.        var newId = Random.Shared.Next(1, 1000);
13.        return Task.FromResult(newId);
14.    }
15. }

```

- **Implement Query Handlers:** Now, we proceed with the implementation of the Query Handler. Following is how we will structure and develop it:

```

1. public class GetProductByIdHandler
2. : IRequestHandler<GetProductByIdQuery, Product>
3. {
4.     public Task<Product> Handle(
5.         GetProductByIdQuery request
6.         , CancellationToken cancellationToken)
7.     {
8.         // Here we would retrieve the product from the
database
9.         // For simplicity, let's return a static product
10.
11.         var product = new Product();
12.         return Task.FromResult(product);
13.     }
14. }

```

At this point, our infrastructure of API is ready. Let us implement two simple minimal APIs able to register a new product and retrieve a product by ID. Obviously, we do not have any database attached, and we will simply put fake data.

This example sets a foundational structure for a CQRS implementation in an ASP.NET Core application, separating command and query responsibilities; we do not have any database attached, enhancing system maintainability and scalability.

Adding the event sourcing

To integrate event sourcing, we need to create an event store and modify the command handlers to emit events instead of directly updating the database state. Additionally, we will use these events to build and maintain our read models.

First, create an Events folder and define a base event class named **BaseEvent** along with specific events related to products as follows:

```

1. public abstract class BaseEvent
2. {
3.     public DateTime Timestamp { get; private set; } = DateTime.
        UtcNow;
4. }

```

Then we will use this **BaseEvent** to specific event for the product creation as follows:

```
1. public class ProductCreatedEvent : BaseEvent
2. {
3.     public int ProductId { get; set; }
4.     public string? Name { get; set; }
5.     public decimal Price { get; set; }
6. }
```

And a second event for product update as follows:

```
1. public class ProductPriceUpdatedEvent : BaseEvent
2. {
3.     public int ProductId { get; set; }
4.     public decimal NewPrice { get; set; }
5. }
```

Event store

An event store is a database designed for storing events as immutable records, allowing systems to reconstruct past states through event sourcing. For simplicity, consider simulating an in-memory event store on your computer, rapidly logging each event to mimic how a real event store chronologically preserves actions within a system.

Following is the implementation:

```
1. public interface IEventStore
2. {
3.     void SaveEvent<T>(T @event) where T : BaseEvent;
4.     IEnumerable<BaseEvent> GetEvents();
5. }
```

The implementation for the in-memory will be as follows:

```
1. public class InMemoryEventStore : IEventStore
2. {
3.     private readonly List<BaseEvent> events
4.         = new List<BaseEvent>();
5.
6.     public void SaveEvent<T>(T @event)
```

```

7.     where T : BaseEvent
8.     => events.Add(@event);
9.
10.    public IEnumerable<BaseEvent> GetEvents()
11.        => events;
12. }

```

This event store captures and preserves every system change, providing a robust foundation for reconstructing historical states.

Modify command handlers

The next step is to adjust the command handlers to emit events rather than directly modifying the state. This approach decouples the state changes from the command processing, enhancing system modularity and scalability. Events can then be processed asynchronously, promoting a more resilient and adaptable architecture while ensuring consistent system updates as follows:

```

1. public class CreateProductHandler
2.     : IRequestHandler<CreateProductCommand, int>
3. {
4.     private readonly IEventStore eventStore;
5.     public CreateProductHandler(IEventStore eventStore)
6.         => this.eventStore = eventStore;
7.     public Task<int> Handle(
8.         CreateProductCommand request
9.         , CancellationToken cancellationToken)
10.    {
11.        var productId = Random.Shared.Next(1, 10000);
12.        var productCreatedEvent = new ProductCreatedEvent
13.        {
14.            ProductId = productId,
15.            Name = request.Name,
16.            Price = request.Price
17.        };
18.        eventStore.SaveEvent(productCreatedEvent);
19.        return Task.FromResult(productId);
20.    }
21. }

```

As we can see in the code, it becomes evident that there is a distinct call to the EventStore, which is prominently made on *line 18*, highlighted by its standalone presence in the script. This emphasizes its importance in the overall functioning of the application.

Similarly, we will implement a specific handler and a specific command to update the product price.

Projections

A projection in event sourcing is a mechanism that listens to and processes events from the event store to update one or more read models. The projection translates the sequence of events into a more readable or useful state, which is then stored in the read model. This updated model provides an optimized query dataset, allowing for quicker and more efficient data retrieval for specific queries without the need to reprocess the entire event log. Essentially, projections help transform raw event data into a structured format that supports faster and more targeted data access. Our simple projection will be crafted as follows:

```
1. public class ProductsProjection
2. {
3.     private readonly Dictionary<int, Product> readModel
4.         = new Dictionary<int, Product> ();
5.
6.     public ProductsProjection(IEventStore eventStore)
7.     {
8.         var events = eventStore.GetEvents();
9.         foreach (var @event in events)
10.        {
11.            Apply(@event);
12.        }
13.    }
14.
15.    private void Apply(BaseEvent @event)
16.    {
17.        switch (@event)
18.        {
19.            case ProductCreatedEvent e:
```

```

20.         readModel[e.ProductId] = new Product
21.     {
22.         Id = e.ProductId,
23.         Name = e.Name,
24.         Price = e.Price
25.     };
26.     break;
27.     case ProductPriceUpdatedEvent e:
28.         readModel.TryGetValue(e.ProductId, out var
product);
29.         if (product is not null)
30.         {
31.             product.Price = e.NewPrice;
32.         }
33.         break;
34.     }
35. }
36. }

```

This setup ensures that all product changes are captured as events and stored in the data store we have chosen. The read model is then built by replaying these events, allowing the system to not only display the current state but also control changes as needed. In any case, modification of archived events must not be allowed. There could be unpredictable consequences.

Visualize results

After implementing CQRS-ES with our products management, in this little paragraph, we will face a simple UI that will allow us to manage in a simple way the API for product management. Let us design a simple product management application where the **Blazor Server** frontend communicates with a backend that utilizes CQRS and event sourcing to handle product information. As usual, we will start with a Blazor WebApp with **Server** as render mode. In this case we will implement the update without relying on SignalR for real-time functionality because we will address this topic in [Chapter 11, SignalR in Real-time Applications](#).

Integrate with backend services

Our backend will be composed of a service class that allows us to call the API to manage our CRUD of products. The class structure will be as follows:

- **HttpServices**

- **AbstractService.cs**
- **IProductService.cs**
- **ProductService.cs**

The **AbstractService** will be able to abstract HTTP calls to our API by implementing methods that can also be reused in other areas. Instead, the **ProductService** will inherit from **AbstractService** and will be able to manage our products via the API made available to our project. Assuming the backend services are implemented, integrate the Blazor Server app with these services by configuring an HTTP Client in **Program.cs**, as follows:

1. `builder.Services.AddHttpClient();`
2. `builder.Services.AddScoped < IProductService, ProductService > ();`

The **AddCors** allows us to call our API, which exposes a localhost address.

1. `builder.Services.AddCors(options =>`
2. `{`
3. `options.AddPolicy("AllowSpecificOrigin",`
4. `builder => builder.WithOrigins("http://localhost:7146")`
5. `.AllowAnyHeader()`
6. `.AllowAnyMethod());`
7. `});`

Then after the `var app = builder.Build();` to activate and use properly the middleware we have to call `app.UseCors();` method.

Build the UI components

Now, we have to show the UI components that are able to interact with the backend. For example, we will craft the following components:

- **Home.razor**: fetches and displays products on the home page.
- **AddProduct.razor**: provides a form for submitting new products.

- **EditProduct.razor**: provides a form for submitting new products.

The **Home.razor** will be as follows:

```

1. <h1>Hello, products!</h1>
2. <p><button @onclick="AddNewProduct">Add new
   product</button></p>
3. @if (products is not null)
4. {
5.     foreach (var product in products)
6.     {
7.         <div style="border:
8.             1px solid black; padding: 10px;
9.             margin: 10px; width:300px">
10.            <p>@product.Id</p>
11.            <p>@product.Name</p>
12.            <p>@product.Price</p>
13.            <p><button
   @onclick="()" = >EditProduct(product)">
14.                Edit
15.            </button>
16.        </p>
17.    </div>
18.    }
19. }
20. @code {
21.     private List<Product>? products;
22.     protected override async Task OnInitializedAsync()
23.     => products = await ProductService.GetProductList();
24.     private void EditProduct(Product product)
25.     => NavManager.NavigateTo($"edit-product/
   {product.Id}");
26.     private void AddNewProduct()
27.     => NavManager.NavigateTo($"add-product");
28. }

```

AddProduct.razor will appear as follows:

```

1. <EditForm Model="@product"
2.     OnValidSubmit="HandleValidSubmit"

```

```

3.     FormName = "AddProductForm" >
4.     <DataAnnotationsValidator />
5.     <ValidationSummary />
6.     <div class = "form-group" >
7.         <label for = "name" > Name: </label>
8.         <InputText id = "name" class = "form-control"
9.             @bind-Value = "product.Name" />
10.    </div>
11.    <div class = "form-group" >
12.        <label for = "price" > Price: </label>
13.        <InputNumber id = "price" class = "form-control"
14.            @bind-Value = "product.Price" />
15.    </div>
16.    <button type = "submit" class = "btn btn-primary" >
17.        Add Product
18.    </button>
19. </EditForm>
20. @code {
21.     private Product product = new Product();
22.     private async Task HandleValidSubmit()
23.     {
24.         await productService.Post(product);
25.         // Reset form after submission
26.         product = new Product();
27.         NavManager.NavigateTo("/");
28.     }
29. }

```

In a similar way, we will craft the **EditProduct.razor** able to modify the selected product price. Now, we just need to call the APIs for each operation. These APIs will correctly query the event store and consolidate all events for each product. This process will ensure that all relevant data is collected and organized, enabling a comprehensive view of each product displayed on the screen.

Implementing CQRS in practice: Summary

This setup provides a simple product management within a Blazor

frontend, using an ASP.NET Core backend with CQRS-ES built in previous paragraphs. This example provides a foundational setup that we can extend and customize based on specific project needs or incorporate more complex behaviors and integrations.

The code can be accessed from the [github link](#).

Conclusion

In conclusion, the integration of CQRS with ES (CQRS-ES) offers a strong architecture for handling complex, scalable applications that require high levels of accountability and performance. By separating the read and write operations, CQRS allows for more tailored optimizations in system performance and usability. Concurrently, ES provides an immutable history of all changes, ensuring every state transition is captured and can be replayed, enhancing audit capabilities and system resilience. This combination not only supports high transaction volumes and complex domain models but also facilitates a more manageable and evolvable system architecture. Implementing CQRS-ES can initially increase complexity, yet the long-term benefits of improved scalability, traceability, and maintainability often outweigh these challenges. For systems demanding strict consistency, detailed audit trails, or where business rules frequently change, CQRS combined with event sourcing represents a compelling solution.

In the next chapter, we will explore the concept of the modular monolith, an approach that allows for maintaining a unified architecture while leveraging the benefits of modularity.

Points to remember

- **CQRS:** It is an architectural pattern that divides writing and reading processes into separate models.
- **Distinct models:** CQRS involves the use of two distinct models: one for handling commands and updating the application state (write model) and another for querying data (read model).
- **Event sourcing:** It is an architectural pattern where changes in application state are stored as a sequence of events.

Multiple choice questions

1. What does CQRS stand for?

- a. Centralized Query Resolution System
- b. Command Query Responsibility Segregation
- c. Complex Query Routing Service
- d. Continuous Query Response System

2. What is the purpose of CQRS?

- a. To combine command and query operations
- b. To separate command and query operations
- c. To reduce system complexity
- d. To increase database performance

3. Which of the following is a benefit of implementing CQRS?

- a. Reduced scalability
- b. Tighter coupling between components
- c. Improved load management
- d. Decreased development flexibility

4. How many models are typically used in CQRS?

- a. One model for both commands and queries
- b. Two models: one for commands and one for queries
- c. Three models: one for commands, one for queries, and one for events
- d. No models are used in CQRS

Answer key

Key terms

- **CQRS:** Command Query Responsibility Segregation.
- **Query operations:** Retrieve and display data without making any changes.
- **Command operation:** Modify data but do not return any results.

- **Event sourcing:** Architectural pattern storing state changes as event sequence.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 10

Modular Monolith

Introduction

In this chapter, we will delve into the core principles of modular monolith architecture, examining how to design and organize modules to achieve both flexibility and performance. In the ever-evolving landscape of software development, understanding how to effectively design and manage modules is crucial for creating scalable, maintainable, and efficient systems. We will explore best practices for communication between modules, ensuring seamless interactions and data flow within our applications. Managing dependencies is another critical focus, where we discuss strategies to handle external libraries and internal module interdependencies. Testing is also emphasized to ensure reliability and quality throughout the development lifecycle. Additionally, we will address the importance of logging and monitoring, providing insights into maintaining visibility and control over your system's health. Finally, we cover deployment and updates, outlining methods to efficiently release new features and improvements while minimizing disruption.

Structure

In this chapter, we will cover the following topics:

- Module design
- Project structure
- Communication between modules
- Dependency management
- Configuration and startup
- Testing
- Logging and monitoring
- Deployment and updates

Objectives

By the end of this chapter, reader will be provided with an in-depth understanding of the modular monolith architectural style in the context of ASP.NET Core. Readers will learn the principles of a modular monolithic application, and to balance the simplicity of a monolith with the flexibility of modular components. The chapter will cover the principles of modularity and the benefits of this approach. Through practical examples, readers will understand how to monitor and measure the performances of an ASP.NET Core project through independent, cohesive modules, enabling testing strategies and scaling. By the end of this chapter, developers will be equipped with the knowledge to effectively utilize modular monoliths, making their applications more robust, manageable, and scalable while retaining the performance advantages of a monolithic deployment.

Module design

A modular monolith is a design pattern where a large application is divided into separate modules, each responsible for a distinct feature or domain. Unlike a traditional monolith, where all components are tightly coupled, a modular monolith emphasizes the separation of concerns and loose coupling within a single executable or deployable unit.

In [Chapter 3, Architecture and Core Components](#), we explored monolithic architecture. The key characteristic of this architecture is

deployed as a single, cohesive package, simplifying the deployment process but potentially complicating scalability and flexibility. In the following figure, we can see how the internal functionalities in a monolith overlap, creating areas of mixed and unclear code:

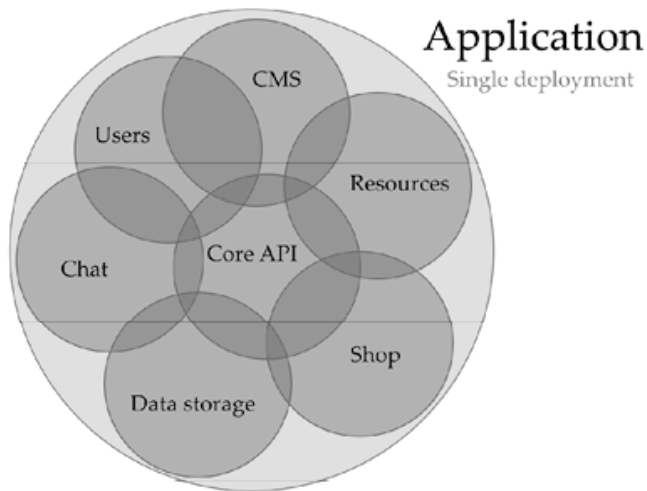


Figure 10.1: *Monolith application*

The downside is the strong interdependence between various parts of the system, creating a tight coupling and making the maintenance and addition of new features difficult, often impossible.

A **modular monolith** refers to a monolithic architecture that incorporates a modular approach in a system design. In a traditional monolithic application, the entire system is implemented as a single monolithic unit. However, with the modular approach, the goal is to structure the monolith into autonomous and well-separated modules, as shown in the following figure:

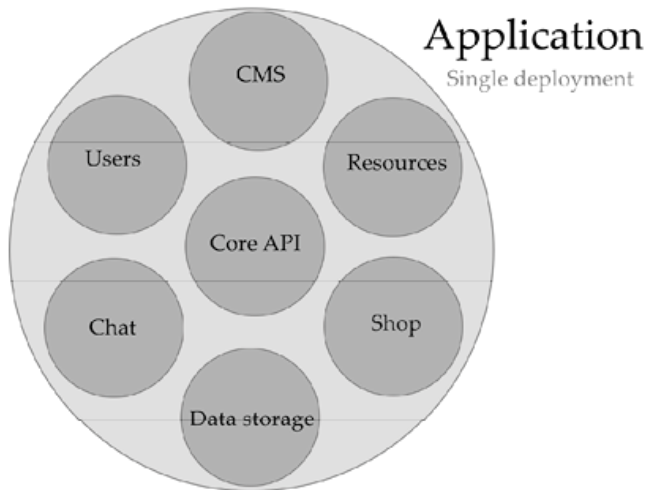


Figure 10.2: *Modular monolith application*

In a modular monolith architecture, we have the concept of microservices but the advantage of a single process.

Key characteristics

Despite its internal modularity, the application is deployed as a single unit, allowing for streamlined deployment processes. Each module has clear and well-defined boundaries, typically enforced by interfaces or APIs, which ensures that the components remain distinct and manageable. This modular structure facilitates independent development and testing of each module, allowing developers to work on different parts of the application simultaneously without interference. The well-defined boundaries also make refactoring more straightforward and less risky. Changes can be made to individual modules without affecting the rest of the application, reducing the likelihood of introducing bugs or causing system-wide issues. Overall, this approach enhances the maintainability and scalability of the application, as each module can evolve independently, and the entire system can adapt easily to new requirements or improvements. This combination of single deployable units and strict module boundaries ensures a robust and efficient development process.

Additionally, a modular monolith offers the advantage of internal communication between modules occurring in-process, eliminating

the network latency typical of microservices and improving the overall performance of the system. Incremental development allows for the addition of new features as separate modules without affecting existing functionality, which facilitates the expansion of the application over time while maintaining management simplicity. In this way, it combines the simplicity of a monolith with the flexibility and efficiency of a modular structure.

Project structure

Since the application is modular, we can choose the architecture and structure shape we want for each single module. The most challenging part remains the defining of the boundaries of each module, which must be clear and impenetrable by other modules. This type of architecture requires a great deal of discipline.

A well-structured modular monolith in ASP.NET Core, in a layered architecture (for example), typically involves the following layers:

- **API layer:** Handles HTTP requests and responses.
- **Application layer:** Contains business logic and application services.
- **Domain layer:** Represents the core domain logic and entities.
- **Infrastructure layer:** Deals with data access, external integrations, and other infrastructure concerns.

We could decide to adopt **Domain-Driven Design (DDD)** for the main structure of our application. For the application module, we should choose **Command Query Responsibility Segregation (CQRS)** as the architecture pattern and other patterns for all other modules.

Following is an example of the Core, Application, and Domain structures:

```
/MyModularMonolith
  /Core
    - Core.Application
    - Core.Domain
    - Core.Infrastructure
    - Core.API
  /Application
```

- Application.Commands
- Application.Queries
- Application.Services
- Application.DTOs

/Domain

- Domain.Entities
- Domain.ValueObjects
- Domain.Aggregates
- Domain.Repositories
- Domain.Services
- Domain.Events

... [CONTINUE]

This structure serves as an example of keeping each module compact and organized, aligning with DDD principles to ensure maintainability and scalability.

Detailed project structure

Our project structure includes not only codes but also additional documents and files for documentation and testing. An example of the structure is as follows:

- **Modules:** Each module has its folder, which contains the following:
 - o **API:** Controllers, models, views.
 - o **Application:** Commands, queries, services, DTOs.
 - o **Domain:** Entities, value objects, aggregates, repositories, domain services, events.
 - o **Infrastructure:** Repository implementations, configurations, DbContext, event handlers.
- **Tests:** Contains the main test project and separate folders for testing each module's components, which are as follows:
 - o **Core.Tests:** Tests for the Core module.
 - o **Module1.Tests:** Tests for Module 1.
 - o **Module2.Tests:** Tests for Module 2.
- **Docs:** Documentation files for various aspects of the project

which are as follows:

- o **Architecture.md**: Overall architecture design and decisions.
- o **DomainModels.md**: Detailed descriptions of domain models and their relationships.
- o **UseCases.md**: Specific use cases and how they are handled within the system.
- o **Requirements.md**: Functional and non-functional requirements of the system.
- **Config**: The configuration files for application settings and logging are as follows:
 - o **appsettings.json**: Configuration for application-specific settings.
 - o **logging.json**: Settings for logging frameworks to control log levels, formats, and destinations.

With this structure, our project will be well-organized, making it easy to locate and improve features.

Principles of modular design

The cornerstone of modular design is the concept of high cohesion and low coupling. High cohesion ensures that all elements within a module are directly related to and necessary for the module's functionality. This principle helps maintain a clear focus within the module, making it easier to develop, test, and maintain. Low coupling, on the other hand, refers to minimizing dependencies between modules. This is crucial for ensuring that changes in one module have minimal impact on others, thereby enhancing the maintainability and scalability of the application.

To achieve these principles in ASP.NET Core, developers should employ DDD techniques where applicable. DDD focuses on defining a clear model for the business domain and aligning the software with the core business concepts. This alignment helps create modules that are logical and reflect real-world business operations.

Defining boundaries and responsibilities

Defining the correct boundaries for modules involves understanding and mapping out the business domain thoroughly. Each module should represent a distinct bounden context (as defined in Domain-Driven Design) of the business with clear responsibilities. For example, in an e-commerce application, typical modules could be *Orders*, *Shop*, and *Customers*. Each of these modules would handle all operations related to their domain, acting somewhat autonomously.

In ASP.NET Core, these boundaries can be physically represented using *Areas*, each corresponding to a specific module. *Areas* help in organizing controllers, views, and other components belonging to a specific functional segment of the application. This physical separation supports logical boundaries and makes the codebase easier to navigate and manage.

Communication between modules

While modules are relatively independent, they often need to communicate with one another. Effective inter-module communication is vital to ensure the application functions as a cohesive whole. In ASP.NET Core, this communication should be designed carefully to avoid creating tight coupling.

We have the following two possibilities of communication:

- Method calls
- Message broker

In method calls communication, each module communicates with others through an API or simply a C# interface. With message broker each module has no idea of the whole application.

Method calls

One communication approach is to use what is commonly named the method call. In programming, a method call is an instruction to execute a predefined function or procedure associated with an object or class. It involves specifying the method's name followed by parentheses containing required arguments. The method processes these arguments and may return a result. The interaction between objects will be strongly coupled, making dependencies difficult to manage for maintenance and evolution. Furthermore,

the structure of the code will be complicated to understand and will require extensive knowledge of the code written (see [Chapter 1, Introduction to ASP.NET Core](#), section *Codebase and complexity*). As we can see in [Figure 10.3](#), communication can quickly become a mess. It will be difficult to understand and maintain:

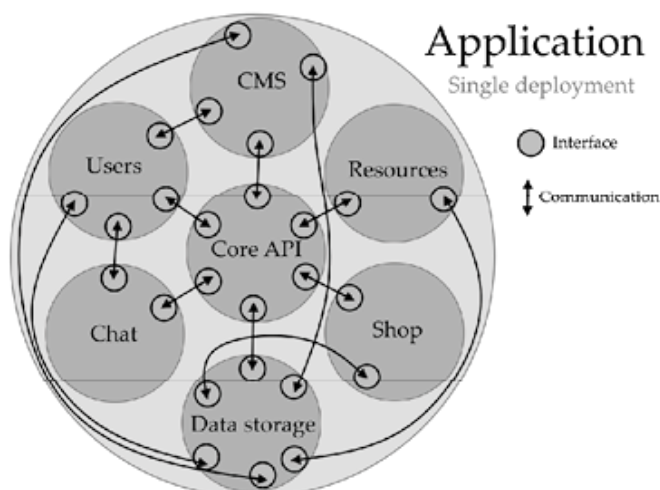


Figure 10.3: Method call communication

Service broker

Communication through a service broker involves an intermediary that facilitates interactions between services, enhancing scalability and flexibility. Services communicate by sending requests to the broker, which routes them to the appropriate service providers, as shown in the following figure:

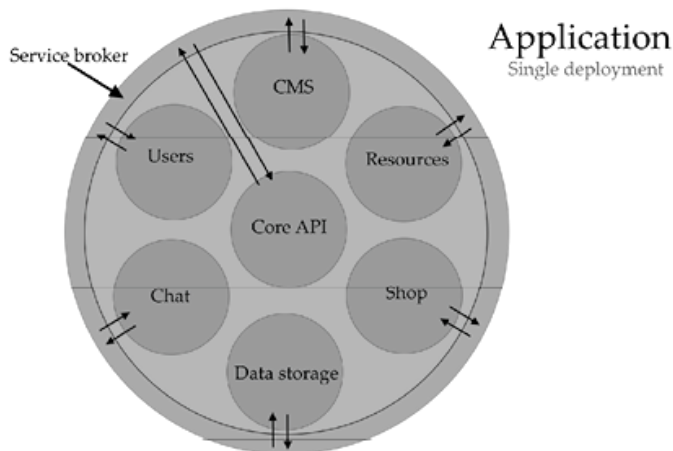


Figure 10.4: Service broker communication

This approach allows modules to communicate without direct dependencies, making the system easier to maintain and evolve. The broker handles tasks like routing, security, and message translation, ensuring efficient and reliable communication. This architecture enhances code organization, scalability, and testability within a single, cohesive application, promoting a clean separation of concerns.

Guidelines

The modular monolith approach aims to strike a balance between the practicality of managing a monolith and the benefits of modularity, providing a compromise between complexity and maintainability. A significant advantage is that switching to a microservices architecture will be much easier.

Shortcuts

One major issue that can occur involves shortcuts between different modules. To avoid shortcuts between modules in a modular monolith architecture, we must enforce strict module boundaries and adhere to principles of modularity and encapsulation. This requires strong discipline from the development team to resist the temptation of quick fixes that violate these boundaries. As developers or architects, we can adopt some strategies to achieve this, ensuring that module boundaries are respected, and the architecture remains clean and maintainable.

Define clear interfaces

Clearly define the interfaces between modules, specifying the allowed interactions and communication protocols. By establishing well-defined boundaries, you prevent modules from directly accessing each other's internals, promoting loose coupling.

Encapsulate module functionality

Each module should encapsulate its own functionality and data, exposing only what is necessary through its interface. Avoid exposing internal implementation details or data structures to other modules, as it can lead to dependencies and shortcuts.

Code reviews

Regular code reviews to identify and prevent shortcuts between modules, the use of static code analysis tools to enforce coding standards and detect boundary violations, along with adopting a zero-tolerance policy towards shortcuts and rule violations, ensure that our codebase will be well-structured, resilient to changes, facilitate the addition of new features, and remain easily maintainable. This approach guarantees a cohesive development process and a robust, scalable application architecture.

Code reviews play a vital role in the software development process, providing an avenue for collaboration, knowledge sharing, and refinement. It is crucial to understand that code reviews *are not personal criticisms*; rather, they offer constructive feedback to improve the quality of the codebase. As we discussed in [Chapter 1, Introduction to ASP.NET Core](#), complexity is subjective and depends on the reader's perception when they review our code. Therefore, if someone describes our code as complex, it is important to take that opinion into consideration. The perception of complexity is more significant the more *educated* the reviewer is about our domain and application codebase.

Maintaining an open-minded approach during code reviews is essential. While it's natural to feel a sense of attachment to our code, being receptive to feedback is fundamental for personal and professional growth. Constructive critiques help identify potential issues, enhance code clarity, and promote adherence to best

practices.

Embracing code reviews with a mindset focused on learning and advancement fosters a positive and cooperative atmosphere within the team. By welcoming feedback, whether positive or negative, individuals demonstrate a commitment to continual enhancement and contribute to a culture of excellence.

Dependency management

Dependency management in a modular monolith involves organizing interactions between modules to ensure maintainability and scalability. The application is divided into distinct modules, each responsible for specific functionalities but still part of a single deployable unit. Clear module boundaries are essential to isolate functionalities, making dependencies easier to manage. Loose coupling allows modules to interact through interfaces or abstractions rather than direct references, reducing the risk of changes in one module affecting others.

Using dependency injection allows modules to declare their dependencies, which are then provided by the application framework, promoting flexibility and testability. Implementing a service broker or mediator pattern within the monolith helps manage communication between modules, ensuring that interactions are decoupled and centrally managed. By focusing on these principles, dependency management in a modular monolith ensures that the application remains maintainable, scalable, and adaptable to changing requirements.

Configuration and startup

In a modular monolith, each module might have its own configuration requirements, such as database connections, API keys, or feature toggles. Each module can have its configuration file, and the main application startup can merge these configurations. For example, a shared configuration service can read and expose configuration settings for each module, ensuring that each module only accesses its settings while maintaining a centralized configuration management approach.

Usually, the built-in configuration in ASP.NET Core is contained in the **appsettings.json** file and loaded in **Program.cs**, which starts the application. However, if all configurations are contained in the **appsettings.json** file, it can become huge and difficult to maintain. A good practice is to divide the configuration into dedicated JSON files and load them in the application. Place the following code snippet at the top of the **Program.cs** file, just after the *using* statements:

```
1. var builder = WebApplication.CreateBuilder(args);
2. builder.Configuration
3.   .AddJsonFile("appsettings.json")
4.   .AddJsonFile($"appsettings.
   {builder.Environment.EnvironmentName}.
   json", optional: true)
5.   .AddJsonFile("module1settings.json", optional: true)
6.   .AddJsonFile("module2settings.json", optional: true)
7.   .AddEnvironmentVariables();
```

In this way, we can have a project folder containing all configurations divided by the module. We could also have configurations within each project's module.

Startup processes in ASP.NET Core involve setting up services and middleware that the application needs to function. In a modular monolith, each module may need to register its services with the dependency injection container. This can be achieved by an extension method of **IServiceCollection** and called from **Program.cs** as:

```
builder.Services.AddOrderModule();
```

This approach maintains a clean separation of concerns, as each module manages its own dependencies without interfering with others.

Middleware components in ASP.NET Core handle tasks such as request logging, authentication, and routing. In a modular monolith, middleware specific to a module can be added conditionally during the application startup. The main application can check for the presence of certain modules and configure the middleware pipeline accordingly.

Routing is also configured during startup. Each module can define

its route mappings, which are then integrated into the main application's routing configuration.

Each module should manage its configurations, dependencies, and middleware, while the main application orchestrates these elements to maintain a cohesive and scalable system.

Testing

Testing in a modular monolith involves ensuring that each module and the overall system works correctly and cohesively. The goal is to maintain high quality and reliability while accommodating the modular architecture. There are three fundamental types of tests that we need to perform in our application. They are as follows:

- **Unit testing:** In this context, it is crucial to focus on individual modules. Each module should have its own set of unit tests to verify the correctness of its functionality. These tests are isolated, ensuring that dependencies are mocked or stubbed out, allowing the module to be tested independently.
- **Integration testing:** It is vital as it verifies that different modules interact correctly. These tests ensure the communication between modules, such as API calls and data exchanges, works as expected. By setting up a controlled environment, integration tests can simulate real interactions, helping to catch issues that unit tests might miss.
- **End-to-end testing:** It covers the entire application workflow, ensuring that all modules work together seamlessly. These tests mimic real user scenarios, testing the complete system from the user interface to the database. This holistic approach helps identify issues in the overall application flow.

In a modular monolith, test automation is essential for maintaining efficiency and consistency. **Continuous integration (CI)** pipelines can run automated tests for each module and application, providing quick feedback and ensuring that changes do not introduce regressions. This approach ensures robust, maintainable, and scalable software. In general, we will dive into testing ASP.NET

Core applications in [Chapter 12, Automated Testing](#).

To achieve our goal of having a good test coverage, one of the best ways to achieve this result is using the V-Model as follows:

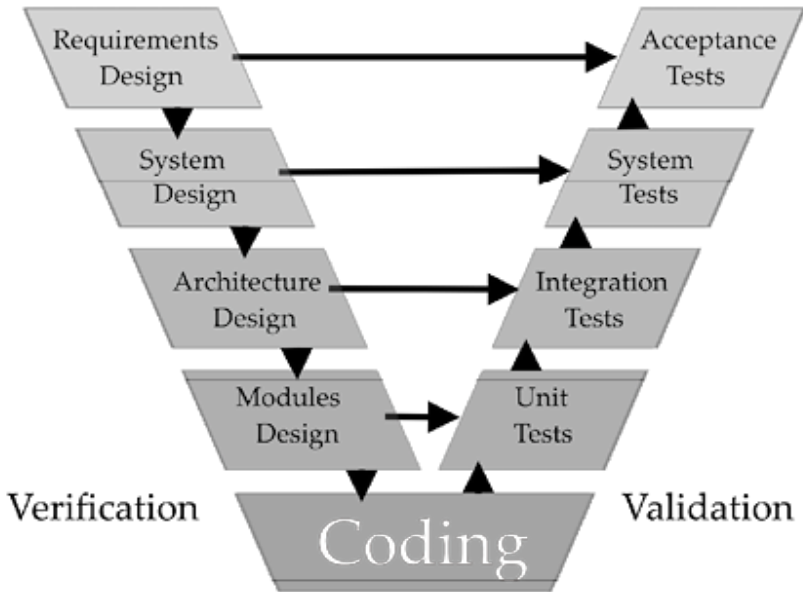


Figure 10.5: V-Model

According to the preceding figure, we can see each phase of application production, starting from requirements to coding and finally to the acceptance of the software. Coding should not merely be a fun activity; writing code is meant to resolve business problems.

Logging and monitoring

In a modular monolith, in ASP.NET Core, logging and monitoring are essential for application health and performance. Logging involves capturing runtime details using the built-in logging framework, which is configurable per module to ensure relevant information is recorded without redundancy. Centralized logging allows logs from all modules to be collected and analyzed together, with structured logging adding contextual data for better traceability.

Monitoring involves tracking metrics like request duration, error rates, and resource usage using tools like *Application Insights* or *OpenTelemetry*. OpenTelemetry is an open-source observability framework for web applications, enabling tracing, metrics, and logging. It provides standardized APIs and SDKs to collect telemetry data, facilitating the monitoring and debugging of distributed systems by integrating with various backend observability tools.

To add **OpenTelemetry** in ASP.NET Core application, add the following **nuget** packages:

```
dotnet add package OpenTelemetry.Extensions.Hosting
dotnet add package OpenTelemetry.Instrumentation.AspNetCore
dotnet add package OpenTelemetry.Instrumentation.Http
dotnet add package OpenTelemetry.Exporter.Console
```

Now, configure it in **Program.cs** as follows:

```
1. builder.Logging.AddSimpleConsole(options =>
2. {
3.     options.TimestampFormat = "[yyyy-MM-dd HH:mm:ss] ";
4. });
5. builder.Services.AddOpenTelemetry()
6.     .WithTracing(tracerProviderBuilder =>
7.     {
8.         tracerProviderBuilder
9.
10.         .SetResourceBuilder(ResourceBuilder.CreateDefault().AddService("WeatherApp"))
11.         .AddAspNetCoreInstrumentation()
12.         .AddHttpClientInstrumentation()
13.         .AddConsoleExporter();
14.     });
```

In this way, we can have a complete trace and metrics in the console. If we would write in a text file, we can use *Serilog*. The following configuration is quite easy:

```
1. Log.Logger = new LoggerConfiguration()
2.     .MinimumLevel.Debug()
3.     .WriteTo.Console(outputTemplate: "[{Timestamp:yyyy-MM-dd HH:mm:ss} {Level:u3}] {Message:l} {NewLine} {Exception}")
4.     .WriteTo.File("logs/log.txt", rollingInterval: RollingInterval.Day, outputTemplate: "[{Timestamp:yyyy-MM-
```

```
dd      HH:mm:ss}      {Level:u3}]      {Message:lj}{NewLine}  
{Exception}")
```

```
5.      .CreateLogger();
```

```
6. builder.Host.UseSerilog();
```

The code can be accessed from the [github link](#).

Deployment and updates

The absolute advantage of a modular monolith is the single deployment. Since the entire application is deployed as a single unit, it simplifies the deployment process, reducing the risk of version conflicts between different modules. One major disadvantage is the difficulty in scaling specific parts of the application. Unlike microservices, where individual services can be scaled independently based on their load, a modular monolith must scale as a whole. Another significant advantage of a modular monolith, aside from the convenience of single deployment, is improved maintainability. By clearly defining boundaries, responsibilities, and communication between modules, avoiding shortcuts for each module, the codebase becomes easier to understand, manage, and update. This strong separation of concerns reduces the complexity of the system, making it simple to locate and fix bugs, implement new features, and refactor existing code.

Conclusion

In conclusion, a modular monolith in ASP.NET Core offers a balanced approach between the simplicity of a monolithic architecture and the scalability of microservices. Dividing the application into distinct modules enhances maintainability, scalability, and testability. Effective management of dependencies, configuration, logging, monitoring, and deployment ensures a robust and efficient system. Leveraging CI/CD pipelines further automates and streamlines the development lifecycle. This approach provides a practical, flexible solution for building and evolving complex applications, allowing teams to achieve high performance and reliability without the overhead of a fully distributed system.

In the next chapter, we will touch with hands a simple communication system from server to client with SignalR, a

powerful Microsoft library for notifications.

Multiple choice questions

1. **What is a key characteristic of a modular monolith?**
 - a. Deployed as multiple independent units
 - b. Deployed as a single unit
 - c. No defined module boundaries
 - d. Always requires a message queue
2. **In ASP.NET Core, where should you configure dependency injection for each module?**
 - a. In each module's Startup.cs file
 - b. In the Startup.cs file of the main API project
 - c. In the extension method of IServiceCollection called from Program.cs
 - d. In the Main.cs file of the main API project
3. **What is a benefit of enforcing strict module boundaries in a modular monolith?**
 - a. Increased coupling between components
 - b. More accessible to introduce breaking changes
 - c. Enhanced maintainability
 - d. Decreased performance
4. **Which practice helps maintain the internal implementation details of a module?**
 - a. Exposing all internal classes to other modules
 - b. Avoiding encapsulation
 - c. Using well-defined interfaces
 - d. Sharing internal implementation details with other modules
5. **What should each module focus on in a modular monolith to have a defined bounded context?**
 - a. Multiple unrelated functionalities

- b. A specific domain or feature
- c. General application logic
- d. Only data access

Answer key

Points to remember

- **Module design:** Ensures each module has a clear responsibility.
- **Project structure:** Organizes project structure to reflect modular architecture.
- **Communication between modules:** Uses interfaces, messages through message broker, and dependency injection for module communication.
- **Dependency management:** Manages dependencies to prevent tight coupling.
- **Deployment and updates:** Plans for independent, backward-compatible module updates.

CHAPTER 11

SignalR in Real-time Web Applications

Introduction

SignalR is a powerful ASP.NET library for adding real-time web functionality, enabling servers to push updates to clients instantly. In .NET 9, SignalR has seen significant enhancements, offering better performance, scalability and integration, particularly with Blazor for dynamic client-side applications.

This chapter introduces the basics of using SignalR in .NET 9, guiding through setting up a SignalR Hub, handling client connections, and broadcasting messages. It also covers advanced messaging techniques such as sending messages to individual clients or groups, which is essential for applications like chat systems and live notifications.

Securing an application that uses SignalR is crucial, and this chapter discusses best practices for implementing robust security measures, ensuring safe and reliable real-time communication. By mastering SignalR in .NET 9, we will be equipped to build responsive and secure real-time web applications that provide a seamless user

experience.

Structure

In this chapter, we will cover the following topics:

- SignalR
- SignalR in .NET 9
- Basic use of SignalR
- Messaging clients
- Securing SignalR application

Objectives

By the end of this chapter, readers will have an understanding of inter-process communication system and the advantages of SignalR in real-time web applications. A key focus will be on gaining insights into how SignalR enhances real-time communication and the various benefits it brings to application performance. Additionally, the chapter will cover techniques and strategies to optimize SignalR communication, enabling the development of more efficient and responsive applications. By the end of this chapter, we will have a comprehensive understanding of SignalR's role in real-time communication and the skills to improve its performance effectively.

SignalR

SignalR is a free, open-source and cross platform library that facilitates the integration of real-time web functionality into applications. This library enables real-time communication by allowing server-side code to push updates to connected clients as soon as they occur. This functionality is crucial in client-server scenarios where immediate updates are necessary, ensuring that changes are promptly reflected on the client side. This real-time capability is essential for applications requiring instantaneous server-to-client communication.

Communication server-client

Before we dive into SignalR functionalities, we have to define exactly what server and clients are. In classic software development, a server is a system that provides resources or services, while clients are systems that request and use these resources. Actions always start from the client, which sends requests to the server to retrieve information or perform operations. The server responds to these requests, managing data and services centrally.

Following figure follows a simple diagram for client-server communication:

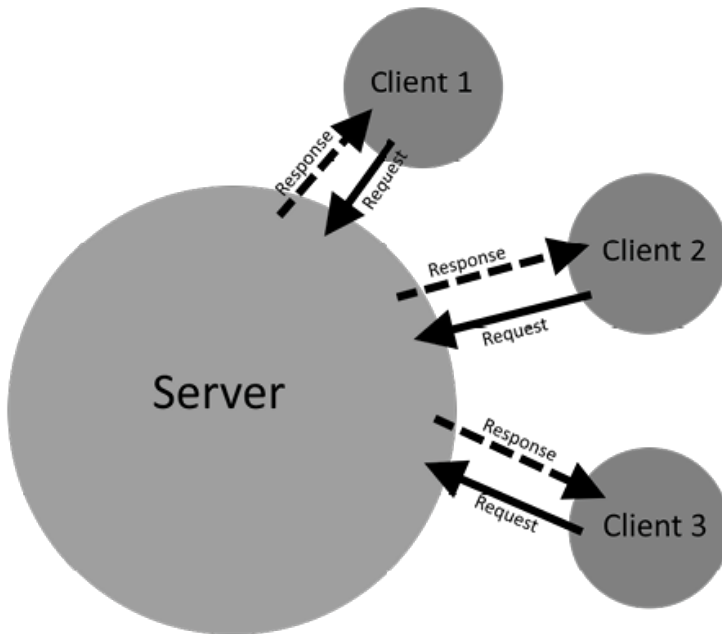


Figure 11.1: Client-server communication

The communication from server to client, or simply software to software, is a modern way to manage information transfer. One of the most challenging scenarios is synchronization between different processes. There are several ways to achieve this objective. Let us look at it in the following section.

Named pipes

Description: Named pipes, also known as **first in, first out (FIFO)**, create a bidirectional channel between two processes using a special

file in the filesystem.

Advantages: Simple to implement, support communication between processes on the same machine.

Disadvantages: Do not scale well for complex or multi-process communications.

Functioning: In C#, **NamedPipeServerStream** and **NamedPipeClientStream** classes are used. The server creates a named pipe and waits for a client connection. Once connected, the server can write data to the pipe, and the client can read from it. This setup allows for a simple and effective way to communicate between processes. Following is an example.

- Server-side implementation
 - o Create a **NamedPipeServerStream** instance with the name of the pipe and direction of communication.
 - o Wait for a client to connect using the **WaitForConnection** method.
 - o Use a **StreamWriter** to send messages through the pipe.

```
1. // Server
2.     using (var pipeServer = new
   NamedPipeServerStream
3.     ("mypipe", PipeDirection.InOut))
4. {
5.     // Wait for a client to connect
6.     pipeServer.WaitForConnection();
7.     using (var writer = new
   StreamWriter(pipeServer))
8.     {
9.         // Write a message to the pipe
10.        writer.WriteLine("Hello from server");
11.        writer.Flush();
12.    }
13. }
```

- Client-side implementation
 - o Create a **NamedPipeClientStream** instance with the name

of the server and pipe.

- o Connect to the server using the **Connect** method.
- o Use a **StreamReader** to read messages from the pipe.

```
1. // Client
2. using (var pipeClient = new NamedPipeClientStream
3.     (".", "mypipe", PipeDirection.InOut))
4. {
5.     // Connect to the server
6.     pipeClient.Connect();
7.     using (var reader = new
8.         StreamReader(pipeClient))
9.     {
10.        // Read and print the message from the pipe
11.        var message = reader.ReadLine();
12.        Console.WriteLine(message);
13.    }
```

Summarise: In this setup, the server waits for a client to connect to the named pipe. Once the client is connected, the server writes a message to the pipe. The client reads this message and displays it. The **NamedPipeServerStream** and **NamedPipeClientStream** classes handle the low-level details of the pipe communication, making it straightforward to implement **inter-process communication (IPC)** in C#. This approach is useful for simple IPC scenarios where processes need to exchange messages in a relatively straightforward manner.

Message queues

Description: Message queues allow processes to send and receive structured messages via a queue managed by the operating system.

Advantages: Support message prioritization, are persistent and flexible.

Disadvantages: Can be complex to manage and require mechanisms for flow control and queue management.

Functioning: In C#, the **System.Messaging** namespace (also known

as MSMQ) is used to interact with message queues. MSMQ provides a way for applications to communicate with each other by sending and receiving messages. The messages are stored in queues and can be retrieved by recipient processes. We can check if a message queue exists and create it if it does not. This ensures the necessary infrastructure for message passing is in place as follows:

```
1. if (!MessageQueue.Exists(@".\Private$\MyQueue"))
2. {
3.     MessageQueue.Create(@".\Private$\MyQueue");
4. }
```

- Sending a message

Create an instance of the **MessageQueue** pointing to the queue and use the **Send** method to send a message. Following is an example that sends a **string** message:

```
1. using (MessageQueue queue = new MessageQueue(@".\Private$\MyQueue"))
2. {
3.     var message = new Message("Hello, World!");
4.     message.Label = "Greeting";
5.     queue.Send(message);
6. }
```

- Receiving a message

Create an instance of the **MessageQueue** class pointing to the queue and use the **Receive** method to retrieve the message. The **Receive** method blocks until a message is available as follows:

```
1. using (var queue = new MessageQueue(@".\Private$\MyQueue"))
2. {
3.     var message = queue.Receive();
4.     message.Formatter = new XmlMessageFormatter
5.         (new String[] { "System.String,mscorlib" });
6.     Console.WriteLine(message.Body.ToString());
7. }
```

Summarise: In this example, the **SendMessage** method sends a

message to the specified queue, and the **ReceiveMessage** method retrieves and displays the message from the queue. This only briefly demonstrates the basic functionality of message queues in C#.

Shared memory

Description: Shared memory allows multiple processes to access a common memory segment, enabling them to share data directly without the overhead of data copying.

Advantages: Very high performance, suitable for large amounts of data.

Disadvantages: Requires synchronization mechanisms to avoid race conditions.

Functioning: In C#, shared memory can be implemented using the **MemoryMappedFile** and **MemoryMappedViewAccessor** classes.

First, create a shared memory segment with **MemoryMappedFile.CreateNew**, specifying a unique name and size. Use a **MemoryMappedViewAccessor** to read and write to the shared memory, enabling efficient inter-process communication. Synchronization mechanisms such as mutexes are necessary to avoid race conditions. Following is an example:

```
1. // Creating shared memory
2. using (var m = MemoryMappedFile.CreateNew("testmap",
    1024))
3. {
4.     using (var accessor = m.CreateViewAccessor())
5.     {
6.         accessor.Write(0, 42); // Writing data
7.     }
8. }
```

A process can access the shared memory segment by opening the existing memory-mapped file with **MemoryMappedFile.OpenExisting** using the same name. A **MemoryMappedViewAccessor** is then created for read or write access. This setup enables efficient inter-process communication. Multiple processes can share data through the common memory segment by using this method. Following is an example:

```

1. // Accessing shared memory
2. using (var m = MemoryMappedFile.OpenExisting("testmap"))
3. {
4.     using (var accessor = m.CreateViewAccessor())
5.     {
6.         int value = accessor.ReadInt32(0); // Reading data
7.         Console.WriteLine(value);
8.     }
9. }
10.

```

Since multiple processes may read from and write to the shared memory concurrently, as we said synchronization mechanisms are essential to prevent race conditions and ensure data consistency. In C#, synchronization primitives such as **Mutex**, **Semaphore** or **Monitor** can be used to coordinate access to the shared memory. These primitives ensure that only one process can access the shared memory at a time, preventing concurrent modifications that could lead to data corruption. Proper synchronization is crucial for maintaining the integrity and consistency of the data being shared among processes, especially in a high-performance, multi-process environment. Following is an example:

```

1. // Creating and using a mutex for synchronization
2. using (var m = MemoryMappedFile.CreateNew("testmap",
3.     1024))
4. {
5.     using (var accessor = m.CreateViewAccessor())
6.     using (var mutex = new Mutex(false, "testmapMutex"))
7.     {
8.         mutex.WaitOne(); // Lock the mutex
9.         accessor.Write(0, 42); // Write data
10.        mutex.ReleaseMutex(); // Release the mutex
11.    }
12. }
13. // Accessing shared memory with mutex
14. using (var m = MemoryMappedFile.OpenExisting("testmap"))
15. {
16.     using (var accessor = m.CreateViewAccessor())

```



```

17. using (var mutex = new Mutex(false, "testmapMutex"))
18. {
19.     mutex.WaitOne(); // Lock the mutex
20.     int value = accessor.ReadInt32(0); // Read data
21.     Console.WriteLine(value);
22.     mutex.ReleaseMutex(); // Release the mutex
23. }
24. }

```

Summarise: In this example, the **Mutex** ensures that only one process at a time can write to or read from the shared memory, thus preventing race conditions and ensuring data consistency. The use of **MemoryMappedFile** and **MemoryMappedViewAccessor** allows for efficient, high-performance interprocess communication, especially suitable for large data transfers.

Server-sent events

Description: **Server-sent events (SSE)** in C# is a standard for establishing a one-way communication channel from the server to the client. It allows the server to push real-time updates to the client over an HTTP connection. SSE is simpler to implement compared to WebSockets and does not require a complex protocol or additional libraries.

Advantages: SSE is straightforward to implement, works over regular HTTP and is supported by most modern browsers.

Disadvantages: SSE is unidirectional (server to client only) and may not be suitable for low-latency applications that require bidirectional communication.

Functioning: To set up SSE in C#, we define an endpoint in our ASP.NET Core application that clients can connect to. The server sends updates to the client by writing to the HTTP response stream.

- Server-side implementation

For the server implementation, create a controller that returns an **IActionResult**. In this method, we define the logic to send updates to the client as follows:

1. // Server Implementation
2. [ApiController]

```

3. [Route("api/[controller]")]
4. public class SseController : ControllerBase
5. {
6.     [HttpGet]
7.     public async Task < IActionResult > Get()
8.     {
9.         Response.Headers.Add("Content-Type", "text/event-
stream");
10.
11.         for (int i = 0; i < 10; i++)
12.         {
13.             await Response.WriteAsync($"data: Message {i}\n
\n");
14.             await Response.Body.FlushAsync();
15.             await Task.Delay(1000);
16.         }
17.
18.         return new EmptyResult();
19.     }
20. }
21.

```

- Client-side implementation

On the client side, you use the **EventSource** API to connect to the SSE endpoint and listen for messages from the server as follows:

```

1. // Client Implementation
2.
3. class Program
4. {
5.     static async Task Main(string[] args)
6.     {
7.         var client = new HttpClient();
8.
9.         var response = await client.GetAsync
10.             ("https://localhost:5001/api/sse"

```

```

11.         , HttpCompletionOption.ResponseHeadersRead);
12.         var stream = await
response.Content.ReadAsStreamAsync();
13.         using var reader = new StreamReader(stream);
14.         while (!reader.EndOfStream)
15.         {
16.             var line = await reader.ReadLineAsync();
17.             if (!string.IsNullOrEmpty(line)
18.                 && line.StartsWith("data: "))
19.             {
20.                 Console.WriteLine
21.                     ($"Message from server:
{line.Substring(6)}");
22.             }
23.         }
24.     }
25. }
26.

```

Summarise: SSE in C# simplifies real-time communication from the server to the client by establishing a unidirectional channel over HTTP. The server sends updates by writing to the HTTP response stream, and the client listens for these updates using the **EventSource** API. This approach is easy to implement and suitable for scenarios where only server-to-client communication is required.

Long polling

Description: There are mainly two types of polling: plain polling and long polling. Plain polling is a technique in which a client periodically sends requests to a server for updates, without waiting for a notification, causing constant traffic. Long polling is a technique where a client holds a connection open until the server has new data, reducing constant requests but increasing wait time before updates arrive.

Long polling in C# is a technique used to achieve real-time communication between the client and server. It involves the client making an HTTP request to the server, which holds the request

open until new data is available to send back to the client. Once the client receives the data, it immediately sends another request, creating a loop that simulates a persistent connection.

Advantages: Long polling works over regular HTTP and is compatible with most environments. It is relatively simple to implement.

Disadvantages: Long polling's drawbacks include increased server load due to open connections, latency if responses are delayed, and scalability challenges under high traffic. Network overhead from reconnections and potential firewall or proxy issues can also impact stability and overall performance.

Functioning: To set up long polling in C#, you define an endpoint in your ASP.NET Core application that handles the polling logic. The server waits for data to become available before responding to the client's request.

- Server-side implementation

For the server implementation, create a controller that returns an **IActionResult**. In this method, you define the logic to wait for and send data to the client as follows:

```
1. [ApiController]
2. [Route("api/[controller]")]
3. public class LongPollingController : ControllerBase
4. {
5.     private static string? message = null;
6.     private static readonly object @lock = new object();
7.     [HttpGet]
8.     public async Task<IActionResult> Get
9.         (Cancellation token cancellationToken)
10.    {
11.        SpinWait.SpinUntil(() => message is not null);
12.        var newMessage = string.Empty;
13.        lock (@lock)
14.        {
15.            newMessage = message;
16.            message = null;
```

```

17.     }
18.     return Ok(newMessage);
19. }
20.
21. [HttpPost]
22. public IActionResult Post([FromBody] string message)
23. {
24.     lock (@lock)
25.     {
26.         LongPollingController.message = message;
27.     }
28.     return Ok();
29. }
30. }

```

To run the long polling server, configure it in the **Program.cs** file of the ASP.NET Core application. Add necessary services and configure endpoints.

- Client-side implementation

On the client side, create a C# console application that uses **HttpClient** to connect to the long polling endpoint and continuously poll for new messages as follows:

```

1. static async Task Main(string[] args)
2. {
3.     var client = new HttpClient();
4.     var pollUrl = "http://localhost:5233/api/longpolling";
5.     var sendMessageUrl = "http://localhost:5233/api/
longpolling";
6.
7.     _ = Task.Run(async () =>
8.     {
9.         while (true)
10.        {
11.            await Task.Delay(5000);
12.            var content = new StringContent
13.                (JsonSerializer.Serialize

```

```

14.                                     ("Hello, Client!
    {DateTime.Now.ToLocalTime()}")
15.                                     , Encoding.UTF8, "application/json");
16.         await client.PostAsync(sendMessageUrl, content);
17.     }
18. });
19.
20. while (true)
21. {
22.     var response = await client.GetAsync(pollUrl);
23.     var responseString = await response.Content
24.         .ReadAsStringAsync();
25.     Console.WriteLine
26.         ("Message from server: {responseString ??
    string.Empty}");
27. }
28. }

```

Summarise: Long polling in C# facilitates real-time communication by keeping the client's HTTP request open until the server has new data to send. The server then responds with the data and the client immediately sends a new request, creating a continuous loop. This method works over regular HTTP and is easy to implement, making it suitable for scenarios where real-time updates are needed but more efficient protocols are not available.

WebSockets

Description: WebSocket provides a full-duplex communication channel over a single TCP connection, allowing for real-time data exchange between client and server.

Advantages: Supports low-latency, real-time communication. The real advantage of WebSocket is that it eliminates HTTP header overhead, a major issue in polling and long polling methods.

Disadvantages: Requires WebSocket support on both client and server. Firewall and proxy compatibility can be an issue.

Functioning: In C#, the **System.Net.WebSockets** namespace is used. The server and client establish a WebSocket connection to

send and receive messages.

- Server-side implementation (ASP.NET Core)

```
1. public class WebSocketHandler
2. {
3.     public async Task HandleWebSocketAsync
4.         (HttpContext context, WebSocket websocket)
5.     {
6.         var buffer = new byte[1024 * 4];
7.         WebSocketReceiveResult result
8.             = await websocket.ReceiveAsync
9.                 (new ArraySegment<byte>(buffer)
10.                  , CancellationToken.None);
11.
12.         while (!result.CloseStatus.HasValue)
13.         {
14.             var serverMsg = Encoding.UTF8.GetBytes
15.                 ($"Server: {DateTime.Now}");
16.             await websocket.SendAsync
17.                 (new ArraySegment<byte>(serverMsg),
18.                  result.MessageType, result.EndOfMessage
19.                  , CancellationToken.None);
20.             result = await websocket.ReceiveAsync
21.                 (new ArraySegment<byte>(buffer)
22.                  , CancellationToken.None);
23.         }
24.         await websocket.CloseAsync
25.             (result.CloseStatus.Value
26.              , result.CloseStatusDescription
27.              , CancellationToken.None);
28.     }
29. }
```

The in the **Program.cs** file we have to add the following endpoint:

```
1. app.UseRouting();
2. app.UseEndpoints(endpoints =>
```

```

3. {
4.     endpoints.Map("/ws", async context =>
5.     {
6.         if (context.WebSockets.IsWebSocketRequest)
7.         {
8.             WebSocket webSocket = await context
9.                 .WebSockets.AcceptWebSocketAsync();
10.            var handler = new WebSocketHandler();
11.            await handler.HandleWebSocketAsync
12.                (context, webSocket);
13.        }
14.        else
15.        {
16.            context.Response.StatusCode = 400;
17.        }
18.    });
19. });

```

- Client-side implementation (console application)

In the client-side we have to use **ClientWebSocket** to establish a WebSocket connection to **ws://localhost:5000/ws**. In the **Main** method of a hypothetical client, the connection is established and two asynchronous tasks, **SendMessages** and **ReceiveMessages** are started to run concurrently.

SendMessages sends a message containing the current date and time every second while the WebSocket connection is open. Meanwhile, **ReceiveMessages** continuously listens for messages from the server and process incoming messages. The program ensures real-time, bidirectional communication over WebSocket using asynchronous methods for efficient execution enabling smooth and simultaneous sending and receiving of messages. Following is an example of a client able to connect and send messages to previous server:

```

1. class Program
2. {
3.
4.     static async Task SendMessages(ClientWebSocket

```



```

WebSocket)
5.  {
6.      while (WebSocket.State == WebSocketState.Open)
7.      {
8.          string message = $"Client: {DateTime.Now}";
9.          var bytes = Encoding.UTF8.GetBytes(message);
10.         await WebSocket.SendAsync
11.             (new ArraySegment<byte>(bytes)
12.              , WebSocketMessageType.Text
13.              , true
14.              , CancellationToken.None);
15.         await Task.Delay(1000);
16.     }
17. }
18.
19. static async Task ReceiveMessages(ClientWebSocket
WebSocket)
20. {
21.     var buffer = new byte[1024 * 4];
22.     while (WebSocket.State == WebSocketState.Open)
23.     {
24.         var result = await WebSocket
25.             .ReceiveAsync(new
ArraySegment<byte>(buffer)
26.                          , CancellationToken.None);
27.         var message = Encoding.UTF8.GetString
28.             (buffer, 0, result.Count);
29.         Console.WriteLine(message);
30.     }
31. }
32. }

```

SignalR overview

Each of the above methods has its own use cases, advantages, and limitations, depending on the specific requirements of the application such as the need for real-time communication,

bandwidth efficiency, implementation complexity, and latency. For instance, WebSockets are ideal for low-latency, bidirectional communication, SSE are excellent for efficient unidirectional updates from the server to the client, and Long polling is effective for maintaining a persistent connection for real-time data retrieval.

At this scope, SignalR manages most of protocols by itself. SignalR primarily uses the WebSocket protocol. WebSocket is known for its efficiency in creating a two-way interactive communication session between the user's browser and a server. This protocol is ideal for scenarios requiring rapid data exchange, such as live chat applications, gaming, financial dashboards, and IoT monitoring. However, we do not need to handle the complexities of WebSocket directly if we use SignalR, as we have seen in the previous example. SignalR abstracts these details, making the integration seamless and developer friendly.

Firstly, SignalR attempts to establish a WebSocket connection due to its high efficiency and low latency. If **WebSocket** is unavailable, SignalR will use SSE, which allows the server to push updates to the client over a single HTTP connection. If SSE is also unavailable, SignalR will fall back to **Long polling**, which simulates a persistent connection by repeatedly polling the server with requests and immediately receiving responses. This ensures real-time functionality even in restrictive network conditions, though with more latency and overhead. SignalR automatically selects the best available protocol for each client's environment.

One of the key strengths of SignalR is that as a developer, we typically do not need to worry about which protocol is being used. SignalR handles the negotiation and selection of the appropriate protocol automatically. However, for specific scenarios or optimizations, you can explicitly specify the desired protocol in the client configuration. Despite this flexibility, the choice of protocol has no impact on how you write your SignalR code. The application code remains consistent and functions the same way, regardless of whether WebSocket, server-sent events or Long polling is used. This consistency simplifies development and ensures that the real-time features of the application are robust and adaptable to various network conditions and client capabilities.

SignalR communication

By abstracting the complexities of bidirectional communication, SignalR enables developers to focus on the core functionality of their applications. At its core, SignalR establishes a persistent connection between the server and the client, allowing messages to be sent and received as events occur. This is achieved through the use of WebSockets when available, falling back to other techniques such as server-sent events and Long polling when necessary. WebSockets provide a low-latency, full-duplex communication channel over a single, long-lived connection, making them ideal for real-time web applications.

SignalR's architecture is built around hubs, which are high-level pipeline models that handle communication between clients and servers. A hub allows clients to call methods on the server and vice versa, with the server managing all active connections and groups of connections. This hub-based approach simplifies the management of client-server interactions, promoting a clean and organized code structure. One of the significant advantages of using SignalR in ASP.NET Core applications is its ability to automatically handle connection management, including reconnection logic and scaling across multiple servers. This is crucial for maintaining the reliability and performance of real-time web applications, especially those with a large number of simultaneous users.

SignalR enhances performance and scalability by integrating with ASP.NET Core's dependency injection and middleware pipeline, allowing us to utilize existing components like authentication, logging, and error handling. This integration ensures consistent and efficient application development.

Furthermore, SignalR offers comprehensive client libraries for platforms like JavaScript and .NET, simplifying the implementation of real-time features and reducing the learning curve. This consistency enables us to build robust, scalable real-time web applications that efficiently handle diverse client environments. Leveraging SignalR, we can create applications that reach a broad audience with reliable, real-time communication, ensuring both flexibility and performance. SignalR's holistic approach to integrating real-time capabilities into ASP.NET Core applications

not only simplify development but also ensures that applications are well-equipped to meet the demands of real-time interactions. By utilizing existing middleware components and supporting multiple transport protocols, SignalR allows us to focus on building feature-rich applications without worrying about the underlying communication mechanisms.

Following figure is a diagram to explain how SignalR works:

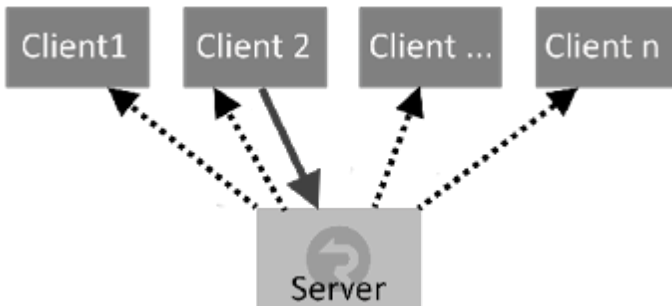


Figure 11.2: *SignalR server communication*

Typical use cases for SignalR include live chat applications, real-time dashboards, collaborative tools, notifications, and any application that requires immediate updates without the need for the client to repeatedly request new data. By providing a simple API for real-time functionality, SignalR significantly reduces the complexity involved in developing responsive and interactive web applications.

SignalR for ASP.NET Core is a powerful tool for building real-time web applications, offering a seamless and efficient way to handle client-server communication. Its integration with ASP.NET Core's infrastructure ensures consistency and efficiency, leveraging existing components like authentication, logging, and error handling. SignalR automatically manages connections and supports multiple transport protocols, making it adaptable to various client environments, including web browsers, desktop applications, and mobile devices. These features make SignalR an essential component for modern web development, enabling the creation of high-performance, scalable and reliable real-time web applications.

Example use cases for SignalR

As a versatile library, SignalR is designed for real-time server-to-client updates and frequent data exchanges, ideal for applications demanding instant communication and data synchronization.

In high-frequency data scenarios, SignalR is invaluable. For gaming, it provides immediate feedback and interaction, ensuring a smooth and engaging experience. Voting or polling applications benefit from real-time updates for accuracy and participant engagement. Auctions use SignalR for dynamic and competitive real-time bidding.

Dashboards and monitoring systems thrive with SignalR. Company performance dashboards, financial market data, instant sales updates, multiplayer game leaderboards and IoT monitoring systems utilize SignalR for live data, aiding timely decisions and enhancing functionality.

In chat applications, SignalR is crucial. Live chat rooms, chatbots, online customer support, real-time shopping assistants, and in-game chat features rely on it for immediate communication, improving user interaction and satisfaction.

For real-time location tracking, SignalR is perfect. Logistics, delivery status, transportation updates, and GPS tracking use it for accurate, timely information, essential for logistics and transportation industries.

Push notifications and real-time broadcasting benefit from SignalR. Social networks, email alerts, game notifications, travel alerts, live audio or video streaming, live captioning, translating and event or news broadcasting use it for seamless, interactive experiences.

In IoT and connected devices, SignalR is vital. Real-time IoT metrics, remote control, and status tracking ensures effective device management and operation. Automation systems benefit from real-time triggers, improving efficiency and responsiveness.

Overall, SignalR enhances the functionality, interactivity and user experience of applications requiring real-time data exchange and communication.

SignalR in .NET9

The latest version of SignalR, introduces several enhancements

designed to improve performance, usability, and security for real-time web applications. Following are some improvements:

- **Stateful reconnects:** SignalR now supports stateful reconnects, allowing clients to resume their connections seamlessly after temporary network disruptions. This feature has been extended to the TypeScript client, enhancing the resilience of real-time web applications.
- **Server timeout and keep-alive intervals:** now we can set server timeouts and keep-alive intervals directly on the **HubConnectionBuilder**. The default values are 30 seconds for **ServerTimeout** and 15 seconds for **KeepAliveInterval**, providing better control over connection management and improving reliability.
- **Circuit activity monitoring:** in server-side Blazor applications using SignalR, now is possible monitor inbound circuit activity with the **CreateInboundActivityHandler** method. This enhancement allows for better tracking and management of real-time connections and interactions.
- **Enhanced logging:** SignalR now includes improved logging capabilities. The new **IHttpLoggingInterceptor** allows for custom logging of request and response details, making it easier to debug and maintain real-time web applications.

These features make SignalR more robust and easier to manage, helping build more reliable and efficient real-time web applications.

Basic use of SignalR

In the following paragraph, we will see a first example of an ASP.NET Core application using .NET 9 and the SignalR protocol for transmission. We will start by setting up the hub and then build a simple application that reacts to events and notifications from the server to the client. In this section, we will see an example where a simple text message is sent from the server to the clients.

Setting up the SignalR Hub

A SignalR Hub in ASP.NET is a server-side component which allows for broadcasting messages to all clients or sending messages to

specific clients. SignalR Hubs simplify the process of implementing real-time features like chat applications, live updates, or notifications by handling connection management, message distribution and client-server communication efficiently. This enables responsive and interactive web applications without the need for complex polling mechanisms.

To start, we need to create a SignalR Hub interface that defines the contract for sending text data to clients. The initial interface should be as follows:

```
1. public interface IHub
2. {
3.     Task SendMessage(object message);
4. }
```

This interface allows any implementing class to send a message. Next, we implement the concrete class as follows:

```
1. public class CommunicationHub : Hub<IHub>
2. {
3.     public async Task Message(object message)
4.         => await Clients.All.SendMessage(message);
5. }
```

This **CommunicationHub** class inherits from **Hub<IHub>** and uses the **SendMessage** method to broadcast messages to all connected clients asynchronously.

Client side SignalR

On the client side, we have to set up a client, based on SignalR **HubConnection**. This class is returned by the **HubConnectionBuilder**, a factory that returns an **HubConnection** according to some parameters in input provided by a Fluent pattern. A simple builder should be made as follows:

```
1. HubConnection hubConnection = new HubConnectionBuilder()
2.     .WithUrl(new Uri("https://localhost:7119/communicationhub"))
3.     .Build();
```

In the above snippet, we can see that an address, corresponding to the hub on our server, is passed. When the **HubConnection** is

created, we have to use it in a client as follows:

```
1. private async Task Client()
2. {
3.     hubConnection.On<NotificationTransport>
4.         (nameof(IHub.SendMessage), message =>
5.         {
6.             Message = message.Message ?? string.Empty;
7.             StateHasChanged();
8.         });
9.     await hubConnection.StartAsync();
10. }
```

As we can see, at *line 3* we can find the implementation of the callback that will be called when a message arrives. In the callback, we have the update of the message variable that will update the UI.

Configuring SignalR at startup

To configure SignalR services to set up the middleware, we will do it in the **Program.cs** file. Following is a simple setup:

```
1. var builder = WebApplication.CreateBuilder(args);
2. builder.Services.AddSignalR();
3. var app = builder.Build();
4. if (app.Environment.IsDevelopment())
5. {
6.     app.UseHsts();
7. }
8. app.UseHttpsRedirection();
9. app.UseStaticFiles();
10. app.UseRouting();
11. app.MapHub<CommunicationHub>("/communicationhub");
12. app.Run();
```

As we can see in the highlighted lines, at *line 2* we add the middleware which manages the SignalR and at *line 11* we configure the hub address. Now, our simple application can work.

Messages clients

Under the hood, SignalR manages connections using a unique

connection ID assigned to each client when the latter asks to connect the first time. This ID helps the server identify and communicate with individual clients. When a client connects, SignalR negotiates the best transport method supported by both the client and server. The connection process involves the client sending a request to the SignalR Hub endpoint, where the server responds with a connection ID and sets up the chosen transport. SignalR maintains a persistent connection between the client and server, allowing for continuous two-way communication. It handles connection lifetime events like reconnections and disconnections automatically.

Messages are exchanged in a JSON or MessagePack format, ensuring efficient data transmission. SignalR's hub API simplifies calling server-side methods from clients and vice versa, making it straightforward to implement real-time features like live updates and notifications in web applications.

Single client communication

SignalR can communicate with all connected clients with broadcast messages, group of clients and single clients. To communicate with a single client in SignalR, we need to use the client's unique connection ID. SignalR allows us to send messages directly to a specific client by referencing this ID. Following steps show how you can achieve this:

1. **Obtain the connection ID:** each client has a unique connection ID that can be accessed when the client connects to the hub. We can store this ID in a database or in-memory collection if we need to refer it later.
2. **Send a message to a specific client:** use the **Clients.Client** method, passing the connection ID of the target client. This method ensures that the message is sent only to the specified client. Here is the method to add to our **ConnectionHub** and obviously to the **IHub** interface:

```
1. public async Task SendMessageToClient
2.   (string connectionId, string message)
3.   => await Clients.Client(connectionId)
4.   .SendMessage(message);
```

3. Client-side implementation: On the client side, we have a method to handle the received messages as usual

By following these steps, we can effectively send messages directly to a single client in SignalR using their unique connection ID.

Group client communication

SignalR groups provide a way to manage and broadcast messages to a subset of connected clients. They are a useful feature for applications requiring selective message delivery, such as chat rooms, private messaging, or segmented notifications. Groups are dynamic and allow clients to join or leave at any time.

In SignalR, a group is identified by a string name. The server can add or remove connections from groups using the **AddToGroupAsync** and **RemoveFromGroupAsync** methods. Once in a group, a client can receive messages sent to that group via the **Clients.Group** method. This mechanism ensures that only the clients within the specified group receive the messages, reducing unnecessary network traffic and improving efficiency.

Groups are managed by the server and do not persist between connections; a client must rejoin groups if it reconnects. This flexibility and ease of management make SignalR groups a powerful tool for real-time, targeted communication in web applications.

The two methods shall be added to interface **IHub** and then implemented in our **CommunicationHub**, as seen before, will have two new methods as follows:

```
1. public async Task AddClientToGroupAsync(string groupName)
2.     => await Groups.AddToGroupAsync(Context.ConnectionId,
    groupName);
3. public async Task RemoveClientToGroupAsync(string
    groupName)
4. {
5.     await Groups.RemoveFromGroupAsync(Context.ConnectionId,
    groupName);
6.     await Clients.Client(Context.ConnectionId)
7.         .SendMessage("Removed from group");
8. }
```

As seen, the two methods are able to add and remove a client to

group and send them the same message.

In conclusion, the server can send a message to a client's group as follows:

```
1. await Clients.Group(groupName).SendMessage(message);
```

Take note that the method **SendMessage** is part of the user defined interface **IHub** as seen in the first example.

Message pack transmission

MessagePack is an alternative protocol in SignalR that can be used instead of the default JSON protocol. MessagePack offers a more efficient option due to its binary serialization format. By encoding data into a compact binary form, MessagePack significantly reduces the size of the transmitted messages. This compression results in several key advantages as follows:

- The reduced data size means that less bandwidth is required for communication, making data transmission faster and more efficient. This is especially important in scenarios with high-frequency messaging or limited network resources.
- Faster data transmission also results in lower latency, ensuring that messages are delivered and processed quickly. This reduction in latency is critical for maintaining the responsiveness of real-time web applications where timely updates are essential.
- The serialization and deserialization processes with MessagePack are quicker compared to JSON. This speed improvement enhances the overall performance of applications, particularly those experiencing high load or dealing with large volumes of data.
- The improved performance also contributes to better scalability, allowing applications to support more users and handle more messages without compromising speed or efficiency.
- SignalR's support for MessagePack is designed to be seamless, requiring minimal changes to existing code to switch from JSON to MessagePack. This ease of integration allows developers to quickly adopt MessagePack and benefit from

its efficiency and performance enhancements, ultimately leading to a more responsive and scalable real-time web application.

Build a basic example

- Create a new Blazor WebApp project as usual. We can use Visual Studio or the .NET **command line interface (CLI)**.
- Add the necessary NuGet packages for SignalR and MessagePack to the server project. As seen in the previous example, we have to add the **nuget** of **SignalR** as follows:

Microsoft.AspNetCore.SignalR.Client

- Furthermore, for the **MessagePack** protocol, we need an additional **nuget** as follows:

Microsoft.AspNetCore.SignalR.Protocols.MessagePack

In the basic example, we configured SignalR services in an ASP.NET application by utilizing the **AddSignalR** method. We can see some additional configurations of this method as follows:

```
1. builder.Services.AddSignalR(  
2.     hubOptions =>  
3.     {  
4.         hubOptions.KeepAliveInterval =  
5.         TimeSpan.FromSeconds(10);  
6.         hubOptions.MaximumReceiveMessageSize = 65_536;  
7.         hubOptions.HandshakeTimeout =  
8.         TimeSpan.FromSeconds(1);  
9.         hubOptions.MaximumParallelInvocationsPerClient = 20;  
10.        hubOptions.EnableDetailedErrors = true;  
11.        hubOptions.StreamBufferCapacity = 150;  
12.    })  
13.    .AddMessagePackProtocol();
```

Within this method, various hub options are set to optimize performance and enhance error handling. In this example we have more configuration of SignalR such as the **KeepAliveInterval**, ensuring regular communication between the server and clients to maintain the connection, the **MaximumReceiveMessageSize**, a **HandshakeTimeout**, the **MaximumParallelInvocationsPerClient**

that by default is set to one but for this application, for example, we set it to 20, allowing multiple concurrent method invocations from a single client. Because the **MaximumParallelInvocationsPerClient** is an int, theoretically this value could be up to 2147483647, that is the max value of **int** in C#.

Enabling **EnableDetailedErrors** provides more comprehensive error information, useful for debugging purposes. Finally, the **StreamBufferCapacity** is set to 150 even if the default is 10. This value sets the limit on how much data can be buffered for client upload streams. It helps manage memory usage and ensures efficient data streaming during client-server interactions. Setting a higher **StreamBufferCapacity** in SignalR can enhance performance by reducing the frequency of I/O operations, leading to more efficient data streaming. However, it also increases memory usage, which can strain systems with limited resources and potentially impact overall performance. Do not abuse of this and be careful handling the above parameters to avoid unexpected behaviors.

In the final line (*line 11*) of the code above, **.AddMessagePackProtocol()** allows SignalR use the MessagePack protocol. By integrating MessagePack, SignalR can transmit data more compactly, reducing the bandwidth usage and improving the performance of real-time web applications, especially in scenarios with high-frequency message updates or limited network bandwidth. This makes it ideal for applications requiring fast, efficient data transfer, such as gaming, financial trading platforms, or real-time monitoring systems. The use of **.AddMessagePackProtocol()** allows our application to switch from the default JSON protocol to MessagePack seamlessly, enhancing the efficiency of client-server communication without significant changes to the existing application logic.

Now, we have to build a **CommunicationHub** class. The code of SignalR Hub is exactly the same of the previous example in paragraph **Basic use of SignalR**. The differences of the use of MessagePack stay in the client code. Adding the client-side logic to connect to the SignalR Hub and use MessagePack, we need as usual an **HubConnection** but adding the **AddMessagePackProtocol()** as follows:

```
1. HubConnection = new HubConnectionBuilder()
```

```

2.         .WithUrl(new Uri("https://localhost:7140/
communicationhub"))
3.     .WithAutomaticReconnect(reconnectionTimeouts)
4.     .AddMessagePackProtocol()
5.     .Build();
6.     _ = HubConnection.On<NotificationTransport>
7.         (nameof(IHub.SendMessage), StringMessage);
8.     await HubConnection.StartAsync();

```

Where **StringMessage** is a function callback. This code snippet creates and starts a SignalR **HubConnection** to a specified URL, with automatic reconnection and MessagePack protocol support. It then subscribes to the **SendMessage** method, receiving **NotificationTransport** object messages and starts the connection.

Add the UI logic

In the UI, we prefer to use a service class to manage API endpoints and SignalR notifications as we have seen in the previous chapters. The service class is used exclusively for communication. This means that if we need a data model, a database connection or an algorithm, we should refer to external classes or libraries, keeping the service class as lean as possible and limiting its responsibility to data transmission. In simple terms, our service class will look like following:

```

1. public class NotificationService : INotificationService
2. {
3.     private readonly string hubEndpoint
4.         = "https://localhost:7140/communicationhub";
5.     private readonly TimeSpan[] reconnectionTimeouts =
6.     {
7.         TimeSpan.FromSeconds(0),
8.         TimeSpan.FromSeconds(30),
9.         TimeSpan.FromSeconds(60),
10.    };
11.    public event Func<string, Task>? OnMessageReceived;
12.    public NotificationService() => _ = InitializeNotifications();
13.    public HubConnection? HubConnection { get; set; }
14.    public async Task InitializeNotifications()
15.    {

```

```

16.     HubConnection = new HubConnectionBuilder()
17.         .WithUrl(new Uri(hubEndpoint))
18.         .WithAutomaticReconnect(reconnectionTimeouts)
19.         .AddMessagePackProtocol()
20.         .Build();
21.     _ = HubConnection.On<NotificationTransport>
22.         (nameof(IHub.SendMessage), StringMessage);
23.     await HubConnection.StartAsync();
24.     await HubConnection.SendAsync
25.         (nameof(IHub.AddClientToGroup), "TIME");
26. }
27. private void StringMessage(NotificationTransport context)
28.     => OnMessageReceived?.Invoke
29.         (context.Message ?? string.Empty);
30. }

```

In our service class, we use the MessagePack protocol, the group management and the callback of SignalR messages. And now, we can open a browser and navigate to **https://localhost:7273** (in our example or the appropriate URL shown in the console of the Blazor app). We should see the Blazor application and be able to send and receive messages using SignalR and MessagePack that will show seconds and milliseconds. In our simple application, we can see the extremely fast refresh rate.

Following is the result of our code:

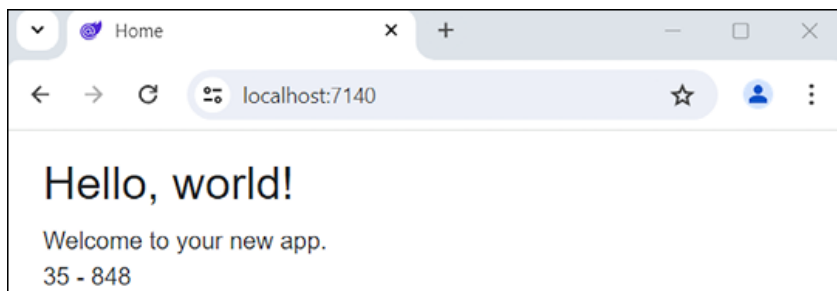


Figure 11.3: Simple application with SignalR

Securing SignalR application

In SignalR for .NET 9, securing communication involves several key

practices to ensure data integrity, confidentiality, and authenticity. Natively SignalR has not security methods built-in, but .NET 9 brings some new features and improvements which can be utilized to secure SignalR applications effectively. The only exception is the MessagePack, which has an evolved security system built-in, but it is only available when MessagePack is properly enabled.

MessagePack security

MessagePack's security, particularly when handling data from untrusted sources, is crucial to prevent attacks such as **Denial of Service (DoS)** or, nowadays, **Distributed Denial of Service (DDoS)**. DDoS attacks can overwhelm a server by sending large volumes of data, leading to memory exhaustion and service disruption. In .NET, MessagePack security is managed through **MessagePackSecurity**, which provides mechanisms to protect against these threats. By using **MessagePackSecurity.UntrustedData**, applications can impose restrictions on the size and structure of incoming data, preventing attackers from sending malicious payloads designed to overload the system. This ensures that the application can handle data safely and continue to operate efficiently even under potential DDoS attacks.

MessagePackSecurity.UntrustedData is a configuration designed to protect the system when processing data from untrusted sources, imposing restrictions both on the maximum size of deserialized objects and on the types of data allowed, reducing risks associated with malformed or malicious data. To implement this security in an ASP.NET Core application using SignalR, you need to configure the MessagePack protocol to apply **UntrustedData**. During the setup of SignalR services, you can add the MessagePack protocol and set **UntrustedData** to ensure that deserialized data is handled securely.

In addition to configuring **UntrustedData**, it is good practice to validate and sanitize received data. While MessagePack provides a level of protection, application-level validation can prevent further vulnerabilities such as the use of cryptographic mechanisms, like digital signatures, to ensure that data has not been tampered with. Implementing proper error handling on both the client and server is essential to manage potential connection or message processing issues. This includes monitoring errors and managing exceptions

and avoiding exposure of stack traces or detailed error messages to users, as they can provide attackers with information about how your application works. It is also important to test the application with malformed data and simulated attacks to ensure the configured security level is effective. This can include load testing to verify resistance against DoS attacks and penetration testing to check robustness against type injection.

Server configuration

In the **Program.cs** file, we set up SignalR and configure it to use the MessagePack protocol with the required security settings. The process begins by creating a new **WebApplication** using **WebApplication.CreateBuilder(args)**, which sets up the application and services. We then add the necessary services to the container, including SignalR and MessagePack. The **AddMessagePackProtocol** method is used to add MessagePack as the serialization protocol. Within this method, we configure **SerializerOptions** through the option **MessagePackSerializerOptions.Standard** and apply **MessagePackSecurity.UntrustedData** to ensure secure deserialization. Additionally, we use **ContractlessStandardResolver.Instance** to simplify serialization without requiring explicit attributes on data types.

Following is the setup in **Program.cs**:

```
1. builder.Services.AddSignalR(  
2.     hubOptions =>  
3.     {  
4.         hubOptions.KeepAliveInterval =  
5.             TimeSpan.FromSeconds(10);  
6.         hubOptions.MaximumReceiveMessageSize = 65_536;  
7.         hubOptions.HandshakeTimeout =  
8.             TimeSpan.FromSeconds(1);  
9.         hubOptions.MaximumParallelInvocationsPerClient = 20;  
10.        hubOptions.EnableDetailedErrors = true;  
11.        hubOptions.StreamBufferCapacity = 150;  
12.    })  
13.    .AddMessagePackProtocol(options =>  
14.    {  
15.        options.SerializerOptions =
```

```

MessagePackSerializerOptions.
    Standard
14.         .WithSecurity(MessagePackSecurity.UntrustedData)
15.         .WithResolver(ContractlessStandardResolver.Instance);
16.     });
17.
18. // ... OMISSIS ...

```

In this configuration, after creating and building the application, we set up middleware to handle exceptions, HTTPS redirection, static files, routing, and authorization. Finally, we map the default controller route and the SignalR Hub before running the application.

Client configuration

On the client side, we similarly configure the SignalR connection and add the MessagePack protocol with the same security settings.

The connection is started asynchronously, and user input is read from the console to send messages to the server.

Following is the complete setup in the client for connection to the server:

```

1. public HubConnection? HubConnection { get; set; }
2. public async Task InitializeNotifications()
3. {
4.     var = new HubConnectionBuilder()
5.         .WithUrl(new Uri("https://localhost:7140/communicationhub"))
6.         .WithAutomaticReconnect(reconnectionTimeouts)
7.         .AddMessagePackProtocol(options =>
8.         {
9.             options.SerializerOptions =
10.                 MessagePackSerializerOptions.Standard
11.                 .WithSecurity(MessagePackSecurity.UntrustedData)
12.                 .WithResolver
13.                 (ContractlessStandardResolver.Instance);
14.         })
15.         .Build();
16. // ... OMISSIS ...

```

The **HubConnectionBuilder** specifies the server URL and adds the MessagePack protocol with the necessary security options. The connection listens for the **SendMessage** event and, in our example, resend the message with an event. In this case our client is in a Service class injected in our page and subscribed to the event.

Custom resolver

Another step for security of an application that uses SignalR transmission is building a custom resolver for MessagePack with specialized formatters. This approach customizes data serialization/deserialization, helping to manage complex types, optimize performance by minimizing data size, and ensure compatibility across versions, tailored to the application's specific requirements. Start by defining custom formatters for the types to serialize and deserialize, implementing **IMessagePackFormatter<T>** to handle the MessagePack format. For example, a custom formatter for a class **Person** type with **Name** and **Age** properties would serialize these properties to a MessagePack array and deserialize them back into a **Person** object. The **Serialize** method writes to the MessagePack writer, while the **Deserialize** method reads from the MessagePack reader, ensuring depth is correctly handled for security. Then, create a custom resolver implementing **IFormatterResolver**. The **GetFormatter<T>** method returns the appropriate formatter, such as **PersonFormatter** for **Person**, or delegates to **StandardResolver.Instance** if the type is unsupported.

Following is an example of a custom formatter and resolver; let us start by defining the **Person** type:

```
1. public class Person
2. {
3.     public string? Name { get; set; }
4.     public int Age { get; set; }
5. }
```

Now, we have to craft a formatter to serialize and deserialize the class **Person**. Following is an example:

```
1. public class PersonFormatter : IMessagePackFormatter<Person>
2. {
3.     public void Serialize(ref MessagePackWriter writer
```

```

4.     , Person value
5.     , MessagePackSerializerOptions options)
6.     {
7.         writer.WriteArrayHeader(2);
8.         writer.Write(value.Name);
9.         writer.Write(value.Age);
10.    }
11.    public Person Deserialize
12.    (ref MessagePackReader reader
13.     , MessagePackSerializerOptions options)
14.    {
15.        options.Security.DepthStep(ref reader);
16.        var count = reader.ReadArrayHeader();
17.        var elements = Enumerable.Range(0, count).Select(_ =>
18.        {
19.            return _ switch
20.            {
21.                0 => reader.ReadString() as object,
22.                1 => reader.ReadInt32() as object,
23.                _ => { reader.Skip(); null }
24.            };
25.        }).ToArray();
26.        var person = new Person
27.        {
28.            Name = elements.ElementAtOrDefault(0)
29.                as string ?? string.Empty,
30.            Age = elements.ElementAtOrDefault(1)
31.                as int? ?? 0
32.        };
33.        return person;
34.    }
35. }
36.

```

Then, implement the custom resolver as follows:

```

1. public class CustomResolver : IFormatterResolver
2. {
3.     public static readonly IFormatterResolver Instance

```

```

4.         = new CustomResolver();
5.     private CustomResolver() { }
6.     public IMessagePackFormatter<T> GetFormatter<T>()
7.         => FormatterCache<T>.formatter;
8.
9.     private static class FormatterCache<T>
10.    {
11.        public static readonly IMessagePackFormatter<T>
formatter;
12.
13.        static FormatterCache()
14.        {
15.            if (typeof(T) == typeof(Person))
16.            {
17.                formatter =
18.                    (IMessagePackFormatter<T>)new
PersonFormatter();
19.            }
20.            else
21.            {
22.                formatter =
23.                    StandardResolver.Instance.GetFormatter<T>();
24.            }
25.        }
26.    }
27. }

```

To use this custom resolver, we need to register it with MessagePack. This registration typically happens in the initialization or configuration section of the application. The custom resolver is included in the **MessagePackSerializerOptions** to ensure it is used whenever MessagePack processes the relevant types.

Following is an example of how to register and use the custom resolver in **Program.cs**:

```

1. builder.Services.AddSignalR(
2.     hubOptions =>
3.     {
4.         hubOptions.KeepAliveInterval =

```

```

    TimeSpan.FromSeconds(10);
5.     hubOptions.MaximumReceiveMessageSize = 65_536;
6.         hubOptions.HandshakeTimeout =
    TimeSpan.FromSeconds(1);
7.     hubOptions.MaximumParallelInvocationsPerClient = 20;
8.     hubOptions.EnableDetailedErrors = true;
9.     hubOptions.StreamBufferCapacity = 150;
10. })
11. .AddMessagePackProtocol(options =>
12. {
13.     options.SerializerOptions
14.         = MessagePackSerializerOptions.Standard
15.         .WithSecurity(MessagePackSecurity.UntrustedData)
16.         .WithResolver(CompositeResolver
17.             .Create(CustomResolver.Instance
18.                 , StandardResolver.Instance));
19. });

```

By defining custom formatters and a resolver, and by registering the resolver with MessagePack, we ensure that the system correctly handles the serialization and deserialization of custom types. This approach provides flexibility and security, particularly when dealing with complex or specific data types.

Conclusion

In conclusion, SignalR offers a powerful framework for building real-time web applications, enhancing user experience with instantaneous data updates and seamless communication. Its ability to handle multiple transport protocols ensures broad compatibility and reliability across different network conditions. By implementing security measures such as TLS for encrypted communication, configuring CORS to restrict access, and incorporating rate limiting to prevent abuse, developers can ensure robust and secure SignalR applications. Despite the initial complexity, the long-term benefits of real-time interaction, improved scalability, and enhanced user engagement make SignalR an invaluable tool for modern web development.

In the next chapter, we will explore the crucial importance of

automated testing, which is essential for ensuring software quality by quickly identifying issues and reducing the risk of bugs reaching the end user.

Points to remember

- **Real-time communication:** SignalR provides real-time bi-directional communication between clients and servers, enabling features like live chat, real-time notifications, and dynamic updates without the need for manual refreshes.
- **Transport protocols:** SignalR automatically selects the best transport protocol available, such as WebSockets, server-sent events, or Long polling, based on client and server capabilities, ensuring optimal performance and compatibility.
- **Security:** SignalR integrates security through the MessagePack protocol, ensuring proper data serialization and deserialization using custom resolvers.

Multiple choice questions

1. What is SignalR primarily used for?

- a. Database management
- b. Real-time web functionality
- c. Image processing
- d. Machine learning

2. In SignalR, how do you send a message to a specific client?

- a. Using `SendMessageToAll()`
- b. Using `Clients.Client(clientId).SendMessage(message)`
- c. Using `SendToClient(clientId)`
- d. Using `BroadcastMessage()`

3. How can you send messages to a group of clients in SignalR?

- a. `Clients.All.SendAsync()`
- b. `Clients.Others.SendAsync()`

- c. `Clients.Group(groupName).SendMessage()`
- d. `Clients.Broadcast(groupName)`

4. Why is it important to secure SignalR applications?

- a. To ensure only authorized users can connect and interact
- b. To optimize performance
- c. To improve user interface design
- d. To simplify development

Answer key

Key terms

- **Hub:** A central class in SignalR that handles client-server communication. It contains methods that can be called by connected clients and can call methods on those clients.
- **Client:** An application or browser connected to the SignalR Hub. Clients can send and receive messages to and from the hub.
- **WebSocket:** A full-duplex communication protocol used by SignalR for real-time data transfer between client and server. It is the preferred transport protocol when available.
- **Groups:** A feature in SignalR that allows you to send messages to specific subsets of connected clients, identified by a group name, facilitating targeted communication.
- **Connection ID:** A unique identifier assigned to each client connection. It is used to send messages to specific clients.
- **Authorization:** The process of ensuring that only authenticated and authorized users can access SignalR Hubs and methods, typically implemented using attributes like `[Authorize]`.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the

Authors:

<https://discord.bpbonline.com>



CHAPTER 12

Automated Testing

Introduction

The world of software development is constantly evolving, so it is crucial to ensure that the applications we create are reliable and efficient. To this, the most robust and safe way is to go through automated tests. In this chapter we will deal with the crucial importance of automated testing. They, today, are the key to obtain high software quality, allowing us to quickly detect problems and significantly reducing the risk of bugs reaching the end customer.

We will examine how to implement a robust battery of automated tests using test methodologies like V-Model. We will see how to integrate these practices into application development using simple examples. We will also look briefly at the methodology of black-box testing, highlighting its advantages, its limitations and the most common implementation strategies.

By the end of this chapter, we will see how to implement tests for a sample application using the testing structure of the V-Model methodology. In addition to the various phases of testing such as unit testing, integration testing, system testing and acceptance testing provided by the V-Model, we will go a little further with a type of testing that are fundamental in ASP.NET Core applications,

load testing. Through these practical examples, we will acquire practical skills in implementing the most common test phases.

Structure

In this chapter, we will cover the following topics:

- Importance of testing
- Types of tests
- Conducting effective tests in ASP.NET Core

Objectives

By the end of this chapter, readers will have an understanding on how to emphasize how important automated testing is to maintain the dependability and effectiveness of our program. We will examine many test approaches, learning about units, integration, end-to-end testing and their distinct roles in the development process. We will demonstrate how to rapidly and successfully build up and run automated tests within an ASP.NET Core application using real-world scenarios. We may improve the overall quality of our software by developing more dependable, efficient and resilient web apps by putting the ideas outlined here into practice.

Importance of testing

As an essential part of contemporary software development, automated testing ensures the efficiency and reliability of applications. The expectations for speed and accuracy in today's competitive and fast-paced technology scene are too great for manual testing to satisfy on its own. The time needed to find and correct errors is greatly decreased by developers using automated testing to quickly and reliably run a wide range of tests. This improves the overall quality of the product and speeds up the development process.

One of the primary benefits of automated testing is its ability to provide immediate feedback. Developers can run automated tests with every code change ensuring that new features or bug fixes do not introduce new issues. This **continuous integration and**

continuous deployment (CI/CD) approach fosters a culture of constant improvement and rapid iteration, essential for maintaining a competitive edge.

In addition, automated tests improve code coverage. Code coverage is a metric that measures how much source code contained in a program is executed (covered) during an automatic test. This helps us identify the untested parts of a code base ensuring that the tests cover the various paths, conditions and branches, which the code proposes improving trying to ensure the quality and reliability of our software. It also allows the execution of complex test scenarios that would be impossible to perform manually ensuring that all aspects of the application are carefully controlled.

Keep in mind, however, that not all code should be tested because some parts offer little test value or it makes no sense to test. Testing everything can lead to excessive maintenance and slow development without significantly improving quality. A code coverage greater than 90% may not be synonymous with quality and greater robustness compared to a lower code coverage as it may not add value to what we are testing. Wide code coverage can also be counterproductive as it can lead to focusing on quantity versus quality, resulting in surface tests that do not capture significant problems. Instead, it is more effective for targeting critical paths, marginal cases, and complex logic. This approach ensures that tests are meaningful and sustainable, providing a better balance between coverage and practical benefits in software quality.

Ultimately, automated testing is not just about finding bugs, It is about building robust, reliable and high-performance software. By integrating automated testing into the development workflow, teams can deliver higher quality products, improve user satisfaction and greater operational efficiency.

Types of tests

There are several types of automatic tests and methodologies. In this section, we will provide an overview of the two most commonly used methodologies.

The methodologies take into account several aspects of the test by grouping the different test steps by target. For example, tests can be

classified into different categories depending on their purpose and the context in which they are applied. Some common examples of different test methodologies include unit tests, integration tests, system tests and acceptance tests, others include regression testing, which ensures that code changes do not introduce new errors, load tests which evaluate system performance under stress and end-to-end tests, that check the entire flow of applications from start to end, and more. Each type of test has a specific role in the software development process and choosing to integrate one instead of the other gives us the opportunity to test one aspect at the expense of another. Test methodologies may vary from the traditional approach of agile waterfalls each with its own advantages and disadvantages. It is not necessary to implement all the methodologies described but it is crucial to adopt those most suitable for our project and purpose. To identify the most appropriate methodologies, it is important to take into account factors such as the complexity of the project, the available budget, delivery times and the experience of the team. The accurate evaluation of these elements will help us to choose the most effective and efficient approach to ensuring the quality of the software.

This section covers some software testing methodologies.

V-Model

In *Chapter 10, Modular Monolith*, we met the V-Model testing in a quick overview. Now let us try to understand a little bit better how it works. The V-Model or Verification and Validation Model is a structured approach to software development that ensures that each development phase is coupled with a corresponding test phase, improving the overall quality and reliability of the final product. By integrating testing activities throughout the development process, this model facilitates early detection of defects and discrepancies between requests and actual implementation.

In the V-Model, development and testing proceed in parallel with each stage of development being checked for design criteria before proceeding. This approach helps to identify problems in advance, reducing the costs and efforts needed to solve problems later in the development cycle. With testing activities each design requirement

is validated and verified through the V-Model which helps to maintain alignment between the project objectives and the software delivered.

Using the V-Model, teams can reach a higher level of discipline and documentation, as each stage requires a review of the requirements and a match in the tests before proceeding. This model is beneficial in projects where requirements and objectives are clear and stable (i.e. not subject to radical change) as it provides a clear and linear path from requirements to delivery.

Overall, the V-Model, in addition to improving communication and collaboration between development and testing teams ensures that the final product meets both functional and non-functional requirements. By integrating testing from the start and then throughout the development process, following the V-Model helps us provide strong control for building robust, high-quality software that is closely aligned with the customer's expectations.

In the following diagram, we can identify the three phases into which this methodology is divided:

- Verification
- Coding
- Validation

Let us see in detail how they work.

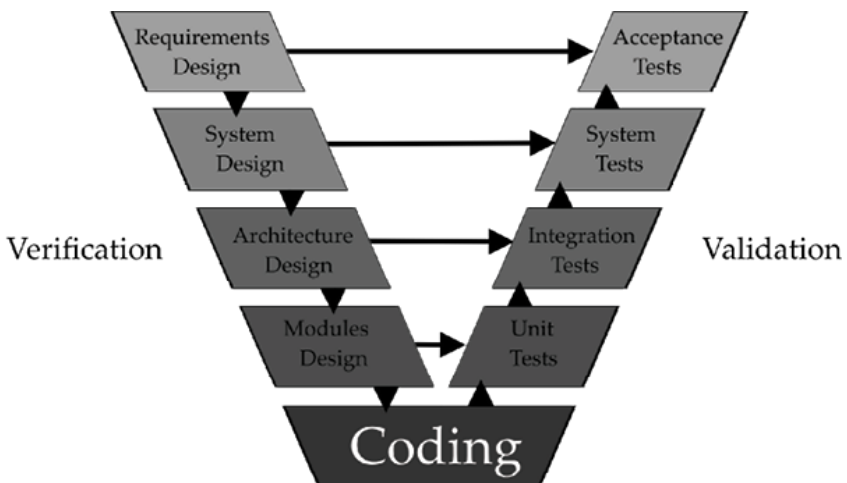


Figure 12.1: V-Model

Verification

The left side of the V in [Figure 12.1](#), which will be followed from top to bottom is the design. The phases of this branch are as follows:

- Requirement design
- System design
- Architecture design
- Modules design

Let us see in an overview of the four phases in the section:

Requirement design

At this stage, system requirements have to be collected and documented. We have already seen in [Chapter 7, Architectural Principles](#) that the documents used for the requirement phase are SAD, SRS, ADR etc. These documents will include both functional requirements, which describe what the system should do and non-functional requirements. (performance, security, usability, etc.). This first step must necessarily ensure as follows:

- **Involvement of stakeholders:** Ensure that all stakeholders are involved to collect complete and accurate requirements.
- **Clear documentation:** Use requirements management tools and maintain concise and clear documentation. In this step the SRS will be written.
- **Review and confirmation:** Review the requirements regularly with stakeholders to ensure that they remain valid and complete.

The various stages of the verification part have a very specific purpose.

This section discusses the four stages of collecting requirements and drawing up documents.

System design

This phase involves the creation of a high-level system design, including the definition of the system architecture, the division into

modules and the specification of interfaces between modules. We can say that in this phase the SAD will be started as follows:

- **Diagrams and patterns:** Unified Modeling Language (UML) diagrams or a C4Model context diagram are used to display an overview of the system architecture.
- **Standards and good practices:** Follow design standards and best practices to ensure consistency and scalability of the future system.
- **Prototyping:** Create prototypes **proof of concept (PoC)** to validate design choices and ensure their feasibility.

Architecture design

At this stage, the technical architecture of the system is designed, defining the main components, their responsibilities and the way they interact. This includes a treaty on technologies to be used as follows:

- **Evaluation of technologies:** Evaluate and choose the most suitable technologies for our project.
- **Scalability and performance:** Design the architecture with particular attention to scalability and performance.
- **Documentation:** We can say that at this stage the SAD document must be drafted using various templates or, as mentioned earlier, the model C4 (container).

Module design

This phase involves the detailed design of each system module. Each module is designed with detailed specifications on how it should be structured and how it ought to interact with other modules including defining data structures, algorithms and interface details as follows:

- **Design models:** Use appropriate design models to address common design problems and ensure best practices.
- **Detailed specifications:** Within the SAD document write detailed specifications for each module including data flow diagrams, interface descriptions and algorithm details (phase component of C4model).

- **Definition of interfaces:** Clearly define the interfaces between the modules to ensure proper communication and integration.
- **Review and validate:** Regularly review and evaluate the design of the modules with colleagues and stakeholders to ensure that they meet system requirements and are feasible in deployment.

The effective completion of the verification phases of V-Model requires good planning, the use of appropriate tools and the continued involvement of stakeholders. The key is to ensure that each step is completed with attention to detail and that continuous verification and validation processes are in place to ensure the final product meets the initial requirements.

Coding

In the requirements design phase, all system requirements including functional and non-functional requirements were collected and documented. The implementation phase translates these requirements into functionality within the ASP.NET core code by creating a software in line with the requirements. Before we start writing the code of our new software, we need to make sure that each requirement is clear to the different team members and well documented, defining the scope of each feature and providing iterative implementation of the features. If that is not the case, it is here that we must return to the previous stages and review all the relevant requirements and documents. During the system design phase, a high-level design of the entire system was specified, identifying the different components and their interactions.

The ASP.NET Core project will be structured on the basis of this design, creating an appropriate structure and ensuring a proper separation of concerns. Using design diagrams to map the system architecture, organize the project into folders and namespaces that reflect the high-level design, and ensure that each component has a clear responsibility are essential tasks. In the architecture design phase, the overall application architecture has been defined, including decisions on technological stacks, frameworks and design models. The implementation of the chosen architecture (as seen in previous chapters) in the ASP.NET Core project involves the

creation of dependency injections, the configuration of middleware and the definition of key architectural components. Critical tasks at this stage are choosing appropriate design models, setting up the project configuration and ensuring that the architecture supports scalability, maintenance and performance. The module design phase envisaged the design of each module in detail, specifying the internal functions and interactions with other modules. It is therefore essential to write the code for each module following every aspect described in the design specifications. If in the SAD we have envisaged the use of vertical slicing, then it is essential that each feature is in a separate project and that it has no cross dependencies except the libraries of common entities of the solution. The next phase of validation includes automatic tests. The implementation of automatic tests is an integral part of each code phase of V-Model. The unit test verifies that each function and module meet the requirements. Integration tests ensure the correct interaction between the different components of the system and the modules produced by our team, so that all communicate with each other as intended. System tests include end-to-end tests to validate that the system meets the requirements required from the user's point of view, while acceptance tests involve stakeholders to test the system in real scenarios or environments very similar to real ones. During implementation it is crucial to develop a well-structured and robust test plan so that you can continuously test the product code to catch problems in advance, then use automated test tools to simplify the verification process.

Following this structured approach, the final product will meet the requirements and maintain high quality, ensuring traceability from requirements to implementation and ultimately to the final software system that will be delivered to the customer.

Validation

The validation testing phase of V-Model is carried out through the use of tests that, however difficult they may be, will have to be automated and automated. It is essential that this phase is extremely detailed to ensure that the software we produce fully meets the specified requirements. This phase acts as a backbone in the life cycle of our application, ensuring that each component and

overall system behaviour is in line with the initial expectations and requirements collected during the verification phase through the SRS document. This approach helps us identify bugs and problems as soon as possible, thereby reducing the costs and efforts required to correct any bugs at a later stage of the development process. As we have already seen earlier, each development phase has a corresponding verification (therefore testing) phase to verify and validate the newly developed software. We will run a series of tests to validate every stage of our project. The unit tests will be the first tests to be performed for the verification of individual units or components of the code. These tests are usually performed by developers. Integration tests test the interaction between different units or components to verify that they work together properly. Then end-to-end tests test the entire system from beginning to end to make sure that the flow of information between various elements of the system works as expected. Finally, the **User Acceptance Tests (UATs)** are performed. They are the final phase of testing in the V-Model and involve end users who verify that the system meets the requirements and expectations. The following is an overview of each test phase:

Unit testing

- **Description:** Tests individual units of code, such as functions or methods in isolation.
- **Advantages:** Identifies bugs early, facilitates refactoring and improves code understanding.
- **Disadvantages:** Can be time-consuming and does not cover integration between units, requires good coverage to be effective.
- **Tools:** xUnit, NUnit, MSTest.

Integration testing

- **Description:** Tests the interaction between different units or components of the software.
- **Advantages:** Identifies issues in component interactions and reduces risks of malfunctions in the integrated system.
- **Disadvantages:** Can be complex to set up, takes more time than unit testing and can be difficult to isolate the cause of

failures.

- **Tools:** xUnit, NUnit, Microsoft.AspNetCore.Mvc.Testing.

End-to-end testing

- **Description:** Tests the entire application flow from start to finish, simulating real-world scenarios.
- **Advantages:** Ensures all parts of the system work correctly together, verifies integration and interoperability between components, covers real usage scenarios.
- **Disadvantages:** Can be complex and costly to implement, requires complete and realistic test environments and can be difficult to identify and isolate the cause of failures.
- **Tools:** Cypress, Selenium, Playwright.

Acceptance testing

- **Description:** Determines if the system meets defined acceptance criteria and is ready for release.
- **Advantages:** Verifies the software meets end-user needs and reduces the risk of customer dissatisfaction.
- **Disadvantages:** Requires a clear set of acceptance criteria, can be influenced by unrealistic expectations and depends on end-user involvement.
- **Tools:** Postman, Specflow.

Black-box testing

Black-box testing is a software test methodology in which the internal structure, design and implementation of the tested element are not known to the tester. This approach focuses exclusively on the input and output data of the software system, examining its functionality without any knowledge of its internal code or architecture.

There are several stages of the black box listed as follows:

- **Functional tests:** The black box test mainly evaluates the functionality of the software in question. Testers provide the agreed inputs and validate the outputs that must be in line with expectations to ensure that the software behaves properly.

- **No internal knowledge required:** Testers do not need to understand the internal operation of the application to be tested. They interact as if they were end users, merely providing input and verifying outputs.
- **Test case design:** Test cases are designed according to the software requirements and specifications.

Advantages

Black-box testing is beneficial because it simulates user interactions to identify issues that real users might encounter. It remains unbiased as testers lack internal knowledge of the code, preventing implementation-related biases. Additionally, it enhances efficiency by allowing testers to create test cases early in the development cycle based on requirements and specifications.

Limitations

Black-box testing has limitations such as limited coverage, as the lack of knowledge of the internal code can result in some parts of the software not being tested comprehensively. Additionally, it may have difficulty identifying specific defects related to the software's internal logic or structure.

Implementation

In black-box testing, it is not always necessary to use the real service. The choice between real and simulated services depends on the testing context. For end-to-end or acceptance testing, using real services ensures operation under production-like conditions, validating that the system meets business requirements and functions as expected. Load testing with real services helps evaluate the application's performance under real conditions, including external interactions.

However, during early development stages or for unit tests, mock services are more suitable. They allow testing without needing access to fully developed services, isolate components for accurate testing and provide increased speed and reliability. This makes them ideal for frequent tests in continuous integration pipelines.

A practical approach combines both real and simulated services.

While unit and integration tests might use mocks for speed and isolation, end-to-end and load tests use real services for comprehensive validation. Configuring the environment to switch between real and mock services helps balance thoroughness, speed and reliability in black-box testing.

Conducting effective tests in ASP.NET Core

When we design a new application from scratch, we consider the things we know at the moment, which may not be complete and sufficient to truly understand the application domain. This will cost us because the specifications will change and consequently, the architecture of our software and the implemented code will also change. It is not possible to manually test the entire application every time a small or large change is made.

In an ASP.NET Core Web API application, we will need a robust and efficient suite of unit tests to avoid unpleasant surprises and regressions during modifications to our application. Unit tests play a key role in ensuring the reliability and stability of our codebase. With thorough testing of individual components and functions, we can immediately identify potential issues and resolve them early in the development process. This proactive approach helps us maintain high code quality and reduces the likelihood of introducing bugs both when implementing new features and when making changes to existing ones.

A comprehensive set of unit tests not only verifies the correctness of our application but also serves as a safety net that gives us the confidence to refactor and optimize our code without fear of breaking existing functionalities. It is necessary to invest time in writing and maintaining tests as they offer significant long-term benefits. However, we always keep in mind that the test writing phase presents us with an extremely challenging task, namely asking ourselves questions about the actual functionality of the part of the code we are testing. Paradoxically, the verification phase will cost us more time investment than the pure writing of the code.

Unit testing

Only for the purpose of this chapter, we can create a simple

standard ASP.NET Core Web API application using Visual Studio 2022 template. The template will add for us a controller that has a simple method **GET** to retrieve the weather simulated information.

Following is **GET** method:

```
1. [HttpGet(Name = "GetWeatherForecast")]
2. public IEnumerable<WeatherForecast> Get()
3. {
4.     return Enumerable.Range(1, 5).Select(index => new
       WeatherForecast
5.     {
6.         Date =
       DateOnly.FromDateTime(DateTime.Now.AddDays(index)),
7.         TemperatureC = Random.Shared.Next(-20, 55),
8.         Summary =
       Summaries[Random.Shared.Next(Summaries.Length)]
9.     })
10. .ToArray();
11. }
```

This method will be the target of our tests for this paragraph.

Arrange-Act-Assert

A prime example of unit tests that can be written are the **AAA** tests. The acronym stands for **Arrange-Act-Assert**, which are the three main phases of a unit test. The **Arrange** phase is responsible for setting up an ideal environment for our test with all necessary components. The **Act** phase will execute the code to be tested and finally, the **Assert** phase will verify that the method produced the expected result. For example, if our sample method returns five items, we would expect a result of five items.

The first part, **Arrange**, will set up the environment where we have to test our method. The second part, the **Act**, will execute the method to test in this protected environment. However, in some cases, the method under test requires external dependencies, such as various objects that are not the focus of this test. The Moq library helps us by ensuring that all objects outside the scope of our unit test respond as expected without affecting the test in our target. The last **A** stands for **Assert** and verify that our test returns the expected

value.

To create a unit test that is handled and recognized by the Visual Studio test engine, we need to use a suitable library. The libraries that can be used with C# and ASP.NET Core are as follows:

- xUnit
- NUnit
- MSTest

These libraries provide the necessary tools and frameworks to write, execute and manage unit tests effectively, ensuring our code behaves as expected. For the purpose of this chapter, we will use xUnit.

Our first simple unit test will look as follows:

```
1. [Fact]
2. public void
3.     Get_ShouldReturnListOfFiveWeatherForecasts_Success()
4. {
5.     // Arrange
6.     var logger = new
       Mock<ILogger<WeatherForecastController>>();
7.     var controller = new
       WeatherForecastController(logger.Object);
8.     // Act
9.     var result = controller.Get();
10.    // Assert
11.    Assert.Equal(5, result.Count());
12. }
```

This code defines a unit test method through the xUnit library to verify that the GET method of our **WeatherForecastController** controller returns a list containing exactly five weather forecasts. We can control any predictable result that our tested method can return against a very precise input. Our tested method will be deterministic if it always returns the same result against the same input. Our unit, in three phases, will arrange preparing everything necessary, an **Act** that will actually call our endpoint, finally, an **Assert** that will check that the result is the expected one.

Particular attention should be paid to conventions on the

designation of test methods. The best practices suggest that the names of the test methods should be descriptive and clear, indicating precisely what is being tested and the expected result (success or fail). Descriptive names help us understand immediately what the purpose of the test is without having to examine the code of it. For example, the previous test is called as follows:

Get_ShouldReturnListOfFiveWeatherForecasts_Success

which clearly indicates that the test is checking a GET method by ensuring that it returns a list of five elements of the weather forecast type and that it is expected to be successful in running the test. This designation convention, in addition to improving readability helps to understand the purpose of the test at a glance. The class name that will contain our tests should have the same name as the class we are going to test with the suffix **Test** so that they can be easily identified and circumscribed. The name of the **WeatherForecastController** class should be **WeatherforecastTests** or **weather ForecastControllerTests**.

Behaviour-Driven Development

The behavioral development approach made with **Behaviour-Driven Development (BDD)**, focuses on describing the behavior of the system in a readable and understandable manner, as close as possible to natural language. The BDD methodology uses structured phrases like **GivenWhenThen** to define test scenarios, in contrast to what we saw in previous section, which used an **Arrange-Act-Assert** structure. This way the description of our system requirements will be made more accessible to all team members, including those who do not have technical roles.

For the implementation in .NET environments we come across frameworks such as **SpecFlow**. It allows you to write tests in a readable format that can be easily converted into executable code. To be able to use it, **SpecFlow**, in the test project in visual studio, we have to define a **function** file. This file is a script in **Gherkin** language and describes the behavior of a software feature in a human-readable format. The purpose of a functionality script is to define the expected behavior of a system functionality from the user's point of view, facilitating clear communication between stakeholders, developers and testers. It serves both as

documentation and as a basis for automated tests. We will be able to test both a single function and a system, we will just describe their behavior in Gherkin language as mentioned before.

To test our controller, **WeatherForecastController**, the **feature** file will be as follows:

Feature: WeatherForecast

Scenario: Get weather forecasts

Given I have a weather forecast controller

When I request the weather forecast

Then I should receive a list of 5 weather forecasts

For convention, this file is a normal **.text** file with **.feature** extension put in a folder named **Features**. Now, we have to define a class that match the Gherkin script. It will look as follows:

```
1. using Chapter12xUnit.Controllers;
2. using Microsoft.Extensions.Logging;
3. using Moq;
4. using TechTalk.SpecFlow;
5.
6. namespace Chapter12xUnit.Tests;
7.
8. [Binding]
9. public class WeatherForecastSteps
10. {
11.     private WeatherForecastController controller;
12.     private IEnumerable<WeatherForecast> result;
13.
14.     [Given(@"I have a weather forecast controller")]
15.     public void GivenIHaveAWeatherForecastController()
16.     {
17.         var logger = new
18.         Mock<ILogger<WeatherForecastController>>();
19.         controller = new
20.         WeatherForecastController(logger.Object);
21.     }
22.
23.     [When(@"I request the weather forecast")]
24.     public void WhenIRequestTheWeatherForecast()
```

```

23.  {
24.      result = controller.Get();
25.  }
26.
27.  [Then(@"I should receive a list of (.*) weather forecasts")]
28.  public void ThenIShouldReceiveAListOfWeatherForecasts(int
count)
29.  {
30.      Assert.Equal(count, result.Count());
31.  }
32. }

```

At this point, we can compile the test project and run the tests from Visual Studio test explorer.

Integration testing

To write an integration test for our ASP.NET Core Web API controller, we can use xUnit along with Microsoft's **Microsoft.AspNetCore.Mvc.Testing** package to create a test project that tests your API endpoint. To set up an integration test we have to install the following packages in our test project if not yet installed:

- **Microsoft.AspNetCore.Mvc.Testing**
- **FluentAssertions**
- **xUnit**
- **Microsoft.NET.Test.Sdk**
- **xunit.runner.visualstudio**

Now, we have to create a new file in the test project **WeatherControllerIntegrationTests.cs** and add the following code:

```

1. public class WeatherControllerIntegrationTests
2.     : IClassFixture<WebApplicationFactory<Program>>
3.     {
4.         private readonly WebApplicationFactory<Program> factory;
5.         public WeatherControllerIntegrationTests
6.             (WebApplicationFactory<Program> factory)
7.             => this.factory = factory;
8.         [Fact]

```

```

9.     public async Task GET_retrieves_weather_forecast()
10.    {
11.        using var client = factory.CreateClient();
12.        var response
13.            = await client.GetAsync("/weatherforecast");
14.        response.StatusCode.Should().Be(HttpStatusCode.OK);
15.    }
16. }

```

This test class inherits from **IClassFixture<WebApplicationFactory<Program>>** to facilitate integration testing. Inheriting from **IClassFixture** ensures that the **WebApplicationFactory<Program>** instance is shared across all tests in the class, promoting efficient resource usage and consistent test setup. The **WebApplicationFactory** class provides a test server for creating an HTTP client to interact with the application. The test method **GET_retrieves_weather_forecast** verifies that a GET request to the **/weatherforecast** endpoint returns an **HTTP 200 OK** status confirming the endpoint's correct functionality.

Another critical point to ensure our test works is to modify the **.csproj** manually. The first line must be as follows:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
```

Without this modification we retrieve an error on *line 13* of the above code.

End-to-end testing

End-to-end (E2E) tests, as we said, verify the functionality of the entire application, ensuring that all components work together as expected. For an ASP.NET Core Web API, E2E tests can involve testing the API endpoints, their interactions and the responses they generate.

Following is an example of E2E tests for the ASP.NET Core Web API using again **xUnit** and **Microsoft.AspNetCore.Mvc.Testing**. For this purpose, we add to our solution a test project from template **xUnit Test Project** then, we will add the following dependencies:

- **Microsoft.AspNetCore.Mvc.Testing**
- **xUnit**

- **Microsoft.NET.Test.Sdk**
- **xunit.runner.visualstudio**

The same as previous integration test, to ensure our test works, we have to modify the **.csproj** manually. The first line will be as follows:

<Project Sdk="Microsoft.NET.Sdk.Web">

At this point, we have to create a new file called **WeatherForecastEndToEndTests.cs** in our new test project and add the following content able to execute the test:

```

1. public class WeatherForecastEndToEndTests
2.     : IClassFixture< WebApplicationFactory< Program> >
3. {
4.     private readonly HttpClient client;
5.
6.     public WeatherForecastEndToEndTests
7.         (WebApplicationFactory< Program> factory)
8.         => client = factory.CreateClient();
9.
10.    [Fact]
11.    public async Task
12.    GetWeatherForecasts_ReturnsExpectedResults()
13.    {
14.        // Act
15.
16.        var response = await client.GetAsync("/
17.        WeatherForecast");
18.
19.        // Assert
20.        response.EnsureSuccessStatusCode();
21.        var forecasts = await response
22.        .Content.ReadFromJsonAsync< IEnumerable< WeatherForecast> > ();
23.        Assert.NotNull(forecasts);
24.        var forecastList = new
25.        List< WeatherForecast> (forecasts);
26.        Assert.Equal(5, forecastList.Count);
27.    }
28. }
```

These end-to-end tests cover the critical paths of our ASP.NET Core API, ensuring that it behaves correctly when all components work together.

Acceptance testing

At this point of our process to produce a wide tests battery, we have to prepare acceptance testing. As we have seen in V-Model paragraph, this type of test ensures that the overall system meets the business requirements and functions correctly from an end-user perspective. In an ASP.NET Core Web API project, acceptance tests typically involve verifying that the API behaves as expected when accessed through its endpoints.

To set up acceptance testing, the steps are the same as previous integration and end-to-end testing projects. For acceptance tests we might also consider using a tool like SpecFlow (seen in unit testing paragraph) for more **Behavior-Driven Development (BDD)** styled tests, but for simplicity, we will stick to xUnit in the following example. Create a new file in your test project called **WeatherForecastAcceptanceTests.cs** and add the following code:

```
1. public class WeatherForecastAcceptanceTests
2.     : IClassFixture<WebApplicationFactory<Program>>
3.     {
4.         private readonly HttpClient client;
5.
6.         public WeatherForecastAcceptanceTests
7.             (WebApplicationFactory<Program> factory)
8.         {
9.             client = factory.WithWebHostBuilder(builder =>
10.             {
11.                 // Customize the server or services if needed
12.                 builder.ConfigureServices(services =>
13.                 {
14.                     // Add any test-specific services or overrides here
15.                 });
16.             }).CreateClient();
17.         }
18.
```

```
19. [Fact]
20. public async Task
   GetWeatherForecasts_ReturnsExpectedResults()
21. {
22.     // Arrange
23.     // Act
24.     var response = await client.GetAsync("/
WeatherForecast");
25.
26.     // Assert
27.     response.EnsureSuccessStatusCode();
28.     var forecasts = await response
29.         .Content.ReadFromJsonAsync
30.             <IEnumerable<WeatherForecast>>();
31.     Assert.NotNull(forecasts);
32.     var forecastList = new
List<WeatherForecast>(forecasts);
33.     Assert.Equal(5, forecastList.Count);
34.
35.     // Additional assertions to verify
36.     // the correctness of the data
37.     foreach (var forecast in forecastList)
38.     {
39.         Assert.InRange(forecast.TemperatureC, -20, 55);
40.         Assert.Contains(forecast.Summary, new[] {
41.             "Freezing"
42.             , "Bracing"
43.             , "Chilly"
44.             , "Cool"
45.             , "Mild"
46.             , "Warm"
47.             , "Balmy"
48.             , "Hot"
49.             , "Sweltering"
50.             , "Scorching"
51.         });
52.     }
53. }
```

```

54.
55.  [Fact]
56.  public async Task
    GetWeatherForecasts_ResponseTimeIsAcceptable()
57.  {
58.      // Arrange
59.      var acceptableResponseTime =
    TimeSpan.FromMilliseconds(2000);
60.
61.      // Act
62.      var startTime = DateTime.UtcNow;
63.      var response = await client.GetAsync("/
    WeatherForecast");
64.      var endTime = DateTime.UtcNow;
65.
66.      // Assert
67.      response.EnsureSuccessStatusCode();
68.
69.      // Check response time is within acceptable limits
70.      var elapsedTime = endTime - startTime;
71.      Assert.True(elapsedTime < acceptableResponseTime
72.          , $"Response time is too slow:
73.          {elapsedTime.TotalMilliseconds}ms");
74.  }
75. }

```

This acceptance test setup is only an example. In a real scenario, we must ensure that the tests confirm the API meets business requirements and performs acceptably under typical usage scenarios.

With SpecFlow

Using SpecFlow for acceptance testing adds a BDD approach to our tests, allowing us to write tests in a more human-readable format. We already introduced SpecFlow in the section on unit testing. Here, SpecFlow integrates with xUnit and can be used to automate the acceptance criteria.

Now, necessary packages to add to our existing **Acceptance.Tests** project will be the following ones:

- **SpecFlow.xUnit**
- **SpecFlow.Tools.MsBuild.Generation**

In the project, we have to add a new file named **specflow.json** that includes the following configuration:

```

1. {
2.   "bindingCulture": {
3.     "language": "en"
4.   },
5.   "language": {
6.     "feature": "en"
7.   },
8.   "unitTestProvider": {
9.     "name": "xunit"
10.  }
11. }
```

Now, we have to create the feature as usual. Create a new folder called **Features** in the test project and add a new file called **WeatherForecast.feature**. This file will contain the Gherkin definition of our test and will be as follows:

```

1. Feature: Weather Forecast API
2.
3.   Scenario: Get Weather Forecasts
4.     Given the weather forecast service is running
5.     When I request the weather forecast
6.     Then I should receive 5 forecasts
7.       And each forecast should have a date, temperature, and
       summary
8.       And the temperature should be between -20 and 55 degrees
9.       And the summary should be one of the predefined summaries
10.
```

At this point we are ready to craft our C# tests creating a new folder called **steps** in test project then adding a new file called **WeatherForecastSteps.cs**. Add the following content as follows:

```

1. [Binding]
2. public class WeatherForecastSteps
3.   : IClassFixture<WebApplicationFactory<Program>>
```

```

4. {
5.     private readonly WebApplicationFactory <Program> factory;
6.     private HttpClient client;
7.     private HttpResponseMessage? response;
8.     private IEnumerable<WeatherForecast>? forecasts;
9.
10.     public WeatherForecastSteps(WebApplicationFactory <Program>
        factory)
11.     {
12.         this.factory = factory;
13.         client = factory.CreateClient();
14.     }
15.
16.     [Given("the weather forecast service is running")]
17.     public void GivenTheWeatherForecastServiceIsRunning()
18.     {
19.         // No setup needed,
20.         // the factory ensures
21.         // the service is running
22.     }
23.
24.     [When("I request the weather forecast")]
25.     public async Task WhenIRequestTheWeatherForecast()
26.     {
27.         response = await client.GetAsync("/WeatherForecast");
28.         if (response is null) return;
29.         if (response.Content is null) return;
30.         forecasts = await response.Content
31.             .ReadFromJsonAsync<IEnumerable<WeatherForecast>>()
32.                 ?? throw new Exception();
33.     }
34.
35.     [Then("I should receive 5 forecasts")]
36.     public void ThenIShouldReceive5Forecasts()
37.     {
38.         Assert.NotNull(forecasts);

```

```

39.     Assert.Equal(5, forecasts.Count());
40. }
41.
42. [Then("each forecast should have a date, temperature, and
summary")]
43.                                     public         void
ThenEachForecastShouldHaveADateTemperatureAndSummary()
44. {
45.     foreach (var forecast in forecasts)
46.     {
47.         Assert.NotNull(forecast.Date);
48.         Assert.NotNull(forecast.Summary);
49.     }
50. }
51.
52. [Then("the temperature should be between -20 and 55
degrees")]
53.                                     public         void
ThenTheTemperatureShouldBeBetweenAndDegrees()
54. {
55.     foreach (var forecast in forecasts)
56.     {
57.         Assert.InRange(forecast.TemperatureC, -20, 55);
58.     }
59. }
60.
61. [Then("the summary should be one of the predefined
summaries")]
62.                                     public         void
ThenTheSummaryShouldBeOneOfThePredefinedSummaries()
63. {
64.     var summaries = new[]{
65.         "Freezing"
66.         , "Bracing"
67.         , "Chilly"
68.         , "Cool"
69.         , "Mild"
70.         , "Warm"

```

```

71.         , "Balmy"
72.         , "Hot"
73.         , "Sweltering"
74.         , "Scorching"
75.     };
76.     foreach (var forecast in forecasts)
77.         Assert.Contains(forecast.Summary, summaries);
78.     }
79. }

```

As we can see, this approach is more verbose than the xUnit approach but we have more control and more flexibility. By following this approach, we can write acceptance tests that ensure our ASP.NET Core application meets the business requirements and behaves correctly from an end-user perspective.

Load testing

Outside the V-Model methodology, we can implement various tests to improve the quality of our software. One of these is certainly load testing.

To write load tests for your ASP.NET Core Web API, we can use a tool like Apache JMeter or a .NET-based tool like NBomber. For simplicity and integration within the .NET ecosystem, we will use NBomber to create and run load tests for this paragraph. For this type of test project, we cannot use the same approach as the others. In this case, we need to use a separate solution and the project will be a console application. We need to add a single NuGet package to our project called **NBomber**.

In the following example, we put all the tests in the **Program.cs** file as top-level statements. The code will look as follows:

```

1. using NBomber.CSharp;
2. using var httpClient = new HttpClient();
3. var scenario = Scenario.Create("weather_forecast_scenario"
4.     , async context =>
5.     {
6.         var response = await httpClient
7.             .GetAsync("https://localhost:7052/weatherforecast");
8.         return response.IsSuccessStatusCode

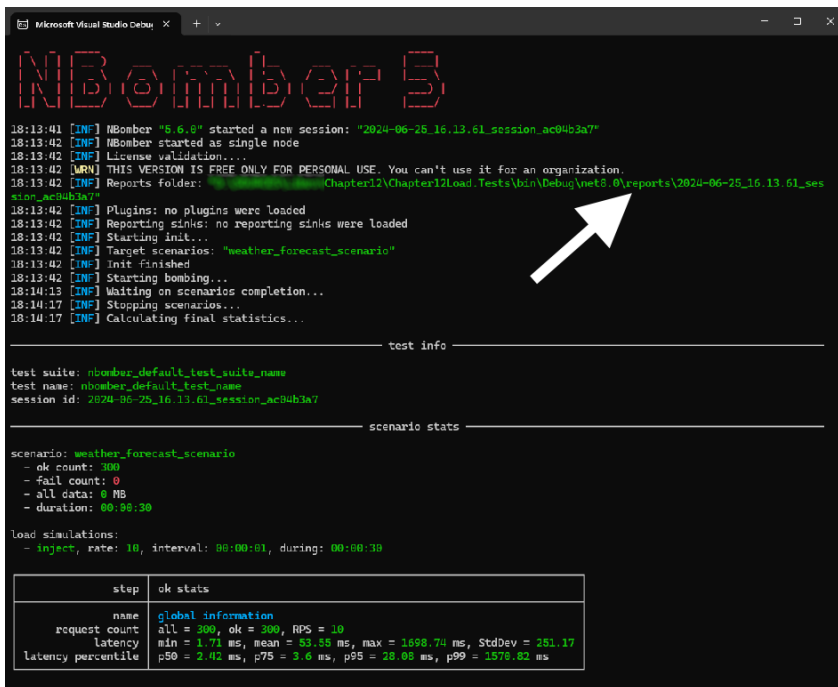
```

```

9.      ? Response.Ok()
10.     : Response.Fail();
11. })
12. .WithoutWarmUp()
13. .WithLoadSimulations(
14.     Simulation.Inject(rate: 10,
15.         interval: TimeSpan.FromSeconds(1),
16.         during: TimeSpan.FromSeconds(30))
17. );
18.
19. NBomberRunner
20.     .RegisterScenarios(scenario)
21.     .Run();

```

The result on the screen will be as shown in the following figure:



```

Microsoft Visual Studio Debu...
NBomber B

18:13:41 [INF] NBomber "5.6.8" started a new session: "2024-06-25.16.13.61_session_ac04b3a7"
18:13:42 [INF] NBomber started as single node
18:13:42 [INF] License validation...
18:13:42 [WARN] THIS VERSION IS FREE ONLY FOR PERSONAL USE. You can't use it for an organization.
18:13:42 [INF] Reports folder: "C:\Users\user\Documents\Chapter12\Chapter12Load.Tests\bin\Debug\net8.0\reports\2024-06-25.16.13.61_session_ac04b3a7"
18:13:42 [INF] Plugins: no plugins were loaded
18:13:42 [INF] Reporting sinks: no reporting sinks were loaded
18:13:42 [INF] Starting init...
18:13:42 [INF] Target scenarios: "weather_forecast_scenario"
18:13:42 [INF] Init finished
18:13:42 [INF] Starting bombing...
18:14:13 [INF] Waiting on scenarios completion...
18:14:17 [INF] Stopping scenarios...
18:14:17 [INF] Calculating final statistics...

----- test info -----
test suite: nbomber_default_test_suite_name
test name: nbomber_default_test_name
session id: 2024-06-25.16.13.61_session_ac04b3a7

----- scenario stats -----

scenario: weather_forecast_scenario
- ok count: 300
- fail count: 0
- all data: 0 MB
- duration: 00:00:30

load simulations:
- inject, rate: 10, interval: 00:00:01, during: 00:00:30

step | ok stats
-----|-----
name | global information
request count | all = 300, ok = 300, RPS = 10
latency | min = 1.71 ms, mean = 93.56 ms, max = 1698.74 ms, StdDev = 251.17
latency percentile | p50 = 2.42 ms, p75 = 3.6 ms, p95 = 28.06 ms, p99 = 1978.82 ms

```

Figure 12.2: NBomber console report

Interpreting results

After running the tests, NBomber will provide a detailed report with

metrics such as:

- **Response time:** Average, minimum and maximum response times.
- **Requests count:** Number of requests made, how ok and how is not ok.
- **Error rate:** Number of failed requests.
- **Latency:** Expressed in milliseconds minimum, maximum and percentile.

The reports will be as shown in the following figure:

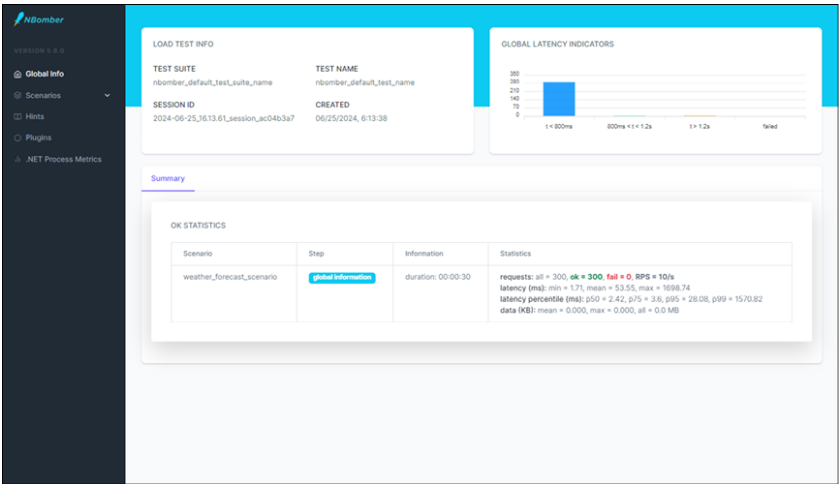


Figure 12.3: NBomber HTML report

These metrics help us understand how our API performs under load and identify potential bottlenecks or issues. This setup allows us to perform load testing on our ASP.NET Core Web API, enabling us to assess its performance and scalability under various load conditions.

Conclusion

In conclusion, this chapter shows us the indispensable role of automated testing in software development, in particular in the ASP.NET Core framework. Automated testing ensures the reliability and efficiency of our applications, enabling us to quickly identify and solve problems while maintaining high software quality standards. Using different testing methodologies such as unit,

integration, end-to-end and black-box testing, we can address different faces of the development process, ensuring that both each component and the system as a whole work properly and as expected.

Planning, introducing and writing automatic tests in the normal development workflow offers significant advantages especially if you plan to insert them into an automatic CI/CD pipeline. Automated tests provide immediate feedback with every code change, ensuring that new features or bug fixes do not present new problems. This promotes a culture of constant improvement and rapid iteration which is essential in today's competitive and fast-paced technological landscape. In addition, automated testing improves the coverage of tests, allowing the execution of complex scenarios that would be virtually impossible to perform manually. This chapter provided a practical guide on configuring and conducting automated tests within ASP.NET Core. By following the strategies discussed like AAA or BDD, we can create more robust, reliable and efficient web applications. Automated testing is not just about bug searching, it allows us to build high-performance software that meets the expectations of customers and users. Finally, integrating automated testing into the development workflow leads to higher quality products, higher user satisfaction and higher overall development efficiency. Software today runs at impressive speeds, but the adoption of automated testing will be an increasingly important part of successful software development, ensuring reliability, scalability and sustainability.

Points to remember

- **Unit tests:** Focus on individual components or functions. They are fast and isolated from other parts of the application.
- **Integration tests:** Test how different components work together, including interactions with external systems like databases or APIs.
- **E2E tests:** Validate the entire workflow of the application, ensuring all components and systems interact correctly.
- **Load tests:** Assess how the application performs under heavy

load or high user traffic.

- **Black-box tests:** Validate the functionality of a software (or service) by testing without knowledge of the internal code structure.

Multiple choice questions

1. What type of test ensures that new changes do not break existing functionality?
 - a. Load test
 - b. Usability test
 - c. Regression test
 - d. Unit test
2. Which tool is used for writing BDD style tests in .NET?
 - a. xUnit
 - b. SpecFlow
 - c. NBomber
 - d. Selenium
3. What is the purpose of WebApplicationFactory in integration testing?
 - a. To mock services
 - b. To provide a user interface
 - c. To create an in-memory server to host the API for testing
 - d. To generate API documentation
4. Which package is necessary for writing load tests with NBomber?
 - a. Microsoft.AspNetCore.Mvc.Testing
 - b. xUnit
 - c. NBomber
 - d. Selenium.WebDriver
5. In a UI test using Selenium, what is the purpose of the

Dispose method?

- a. To set up test data
 - b. To clean up and quit the WebDriver after tests
 - c. To initialize the WebDriver
 - d. To verify test results
- 6. Which HTTP status code indicates that an endpoint requires authentication but no token is provided?**
- a. 403 Forbidden
 - b. 404 Not Found
 - c. 500 Internal Server Error
 - d. 401 Unauthorized
- 7. What is an essential package for creating UI tests with Selenium in a .NET project?**
- a. Microsoft.NET.Test.Sdk
 - b. Selenium.WebDriver
 - c. NBomber
 - d. Microsoft.AspNetCore.Mvc.Testing
- 8. Which type of test would you use to simulate a large number of users accessing your API simultaneously?**
- a. Load test
 - b. Unit test
 - c. Usability test
 - d. Integration test
- 9. Which tool is commonly used for end-to-end testing of web applications, simulating real user interactions?**
- a. NBomber
 - b. Selenium
 - c. SpecFlow
 - d. Swagger

Answer key

Key terms

- **Automated testing:** Essential for maintaining software quality, enabling quick identification and fixing of issues, and minimizing defects in production.
- **Unit testing:** Verifies individual components of the software.
- **Integration testing:** Evaluates interactions between different components of the system.
- **E2E testing:** Assesses the system's overall functionality by testing complete workflows.
- **Black-box testing:** Tests the software without knowing the internal code structure.
- **Test coverage:** Measure of how much of the application's functionality is tested.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 13

Security in ASP.NET Core

Introduction

In today's digital world, ensuring robust web application security is crucial to protecting user data and maintaining online integrity. This chapter explores essential best practices for the security of an ASP.NET Core application, providing insight into effective measures to protect systems. We will see the importance of authorization management, an essential part of a web application to control user access and prevent unauthorized access and activity. Furthermore, we cannot talk about authorization without mentioning the authentication. In addition, we will emphasize the need to use HTTPS and SSL/TLS certificates to encrypt data transmissions, thus ensuring greater security for communication between server and client. By implementing these strategies, we will be able to improve cybersecurity, mitigate risks and promote a secure digital environment for their operations and users.

Structure

In the chapter, we will cover the following topics:

- Best practices for backend security
- Confidentiality, integrity and availability
- Authorization management
- Usage of HTTPS and SSL/TLS certificates

Objectives

By the end of this chapter, readers are supposed to be purely practical and cover transparent information about securing an ASP.NET Core application. We will describe effective permission management for giving access only to system resources that are provided with authorization to do so. We will also discuss how data transferred between server and client is secured through HTTPS and SSL/TLS certificates against potential interceptions via encryption. The chapter eventually offers tangible strategies for strengthening overall application security for the reduction of vulnerabilities and risk, the critical practices in creating secured and reliable application environments for users.

Best practices for backend security

In this section, we will explore best practices to ensure the security of our ASP.NET Core applications. Security is a very important aspect of any web application to be handled more and more carefully. ASP.NET Core provides a variety of tools and features to help us protect our application at least from the most common and known threats. To achieve that, there are two main ways of working to guarantee the security in web applications. The first is resumed by the acronym C-I-A that guarantees, if the three points are respected, that data managed from our server are protected. On the other hand, there is the **golden standard** or more simply **the three Au's**. These guidelines contain three principles that are fundamental to protecting computer systems and sensitive information from unauthorized access and other threats.

Confidentiality, integrity and availability

The acronym C-I-A, in the context of information security, stands for confidentiality, integrity and availability. Each term stands as following:

- **Confidentiality:** This principle focuses on ensuring that information is accessible only to those who are authorized to access it. Confidentiality is achieved through the use of access control and encryption mechanisms which prevent unauthorized persons from viewing or intercepting sensitive data.
- **Integrity:** Integrity ensures that the information is accurate and complete and that it has not been unauthorized. This principle is ensured through checks such as digital signatures, checksums and hashing protocols which help to detect any unauthorized changes to data.
- **Availability:** Availability ensures that information and systems are available for use when needed. This implies that information systems are operational and accessible, protecting against threats such as DDoS attacks, hardware crashes or power outages through redundancy measures, backups and operational continuity plans.

The C-I-A model represents the three fundamental pillars of information security, each of which is essential for protecting data and IT systems from various types of threats and ensuring proper functioning and trust in information systems.

The three Au's

Au is the chemical symbol for gold in the periodic table of elements. The use of **the three Au's (Authentication, authorization, audit)** in cybersecurity is a play on the chemical symbol for gold, **Au** (from the latin *Aurum*), which is the syllable that starts all three words. Given the presence of this syllable in all three words, it was thought that there could be a correspondence between these three principles and the fact that together they could be defined as the **gold standard** or the highest level of excellence in information protection. Let us see more in depth how these three principles work and how (in broad terms) we can implement them, as follows:

- **Authentication:** It is the process of verifying the identity of a

user, device or system to ensure that only authorized users can access specific resources. Common methods include passwords, two-factor authentication, biometrics and digital certificates. For example, we enter a password to access our email account, proving our identity as the account owner.

- **Authorization:** It determines the permissions and privileges an authenticated user has within a system, controlling access to resources and actions based on predefined roles and policies. After authentication, the system checks which resources the user can access and what operations they can perform. Methods include **role-based access control (RBAC)**, where users are assigned specific roles with set permissions, **attribute-based access control (ABAC)**, where access decisions are based on user attributes, requested actions, resources and environment and **access control lists (ACLs)**, which specify which users or groups can access certain resources. For example, an accounting department employee may have access to financial reports, while an IT department employee does not.
- **Auditing:** It is the process of recording and reviewing user and system activities to monitor, track and verify actions for detecting suspicious behavior, ensuring regulatory compliance and conducting investigations in case of security breaches. Key components include system logs, which provide detailed records of events like access attempts, data modifications and resource usage, log analysis, which involves reviewing and analyzing logs to identify anomalous or suspicious activities and audit reports, which summarize activities and events for security personnel or auditors. For example, a security audit might reveal that a user account accessed sensitive data outside of working hours, indicating a possible account compromise.

Pay attention to the fact that **authorization** and **authentication** in common slang are abbreviated by **AuthN** for authentication and **AuthZ** for authorization. Normally, these terms are used in written language as informal conversation. Sometimes, they appear in the common language spoken by IT people but it is rare at the moment. This kind of slang will be ever more present because everyone

knows the meaning and is shorter to use.

We will start to see authorization and authentication later in this chapter in the section *Authorization management*.

Securing configuration and secrets

Another important point to guarantee the application security is how to secure properly the configuration files. In modern applications, configuration files often contain confidential information such as connection strings and API keys. Protecting these files from unauthorized access is essential. In this section, we will describe a couple of common practices for this type of protection, including the secret manager tool or Azure key vault.

Protecting configuration files

In modern applications, configuration files often contain confidential information such as connection strings and API keys. Protecting these files from unauthorized access is essential. In this section, we will describe a couple of common practices for this type of protection as follows:

Encrypted configuration files

For example, we could use encryption to protect configuration files like **appsettings.json**. We can encrypt the contents of the configuration file using encryption tools or libraries installed from **nuget** package. At runtime, the application will decrypt the content to access sensitive information. This ensures that even if the file is compromised, the encrypted data inside will remain protected. In ASP.NET Core, we can use the **Data Protection API (DPAPI)** to encrypt and decrypt our configuration files. The encryption should be made in a separate class with a method similar to the following:

```
1. public void EncryptFile()
2. {
3.     var provider = DataProtectionProvider.Create("MyApp");
4.     var protector = provider.CreateProtector
5.         ("ConfigurationProtector");
6.     var plainText = File.ReadAllText("appsettings.json");
7.     var encryptedText = protector.Protect(plainText);
8.     File.WriteAllText("appsettings.encrypted.json", encryptedText);
```

```
9. }
```

At runtime, at startup of the application, we will decrypt and read the file in **Program.cs** file as follows:

```
1. var provider = DataProtectionProvider.Create("MyApp");
2.         var protector =
   provider.CreateProtector("ConfigurationProtector");
3.         var encryptedConfig =
   File.ReadAllText("appsettings.encrypted.json");
4. var decryptedConfig = protector.Unprotect(encryptedConfig);
5.     builder.Configuration.AddJsonStream(new MemoryStream
6.     (System.Text.Encoding.UTF8.GetBytes(decryptedConfig)));
7. builder.Services.AddDataProtection();
8. builder.Services.AddControllersWithViews();
```

Role-based access control

We can implement stricter access controls to set limits on who can read or edit configuration files. Using role-based access control mechanisms can be a good solution to ensure that only authorized users or processes can access them. For example, we can configure file system permissions so that only the user of the application process has access, effectively preventing unauthorized access. In **Program.cs** we can set the access to the controller able to read config files as follows:

```
1. builder.Services.AddAuthorization(options =>
2. {
3.     options.AddPolicy("ConfigAccess",
4.         policy => policy.RequireRole("ConfigManager"));
5. });
```

In [Chapter 14, Securing Web Applications Effectively](#), we will see in detail how to set properly the access to the controller with effective roles.

Dynamic configuration from a secure configuration service

We can also configure our application to dynamically retrieve sensitive configurations from a secure configuration service. This approach allows the centralized management of configurations and the application of additional security controls to ensure their security. For example, we could use secure configuration services

such as HashiCorp Vault or AWS Secrets Manager. These services offer advanced features for secure configuration and key management, such as automatic key rotation and access control for thin grains.

Encrypting sensitive data

Also, about protection of confidential configuration data, we can encrypt sensitive data both when they are written and during transport to prevent unauthorized access. By using strong encryption algorithms and managing encrypting keys securely, we will have a robust application preventing attacks through our secret keys. Implementing TLS/SSL for data in transit and encrypting writing on databases and storage devices protects us data in rest, ensuring complete security.

Data protection API

In previous section, in encrypted configuration files, we encountered the data protection API to encrypt sensitive data in **appsettings.json** file but we can encrypt many other files such as cookies and form fields, ensuring confidentiality and integrity. The API provides cryptographic services to protect data by encrypting it before storage or transmission, making it unreadable to unauthorized users, thus enhancing overall application security. In the following code, a cookie content is encrypted and decrypted:

```
1. app.MapGet("/get-cookie-content", async context =>
2. {
3.     var protector =
4.     app.Services.GetDataProtector("CookieProtector");
5.     if (!context.Request.Cookies.TryGetValue
6.     ("MyProtectedCookie", out var encryptedCookie))
7.     {
8.         // Encrypting the data
9.         var sensitiveData = "Sensitive Cookie Data";
10.        var encryptedData = protector.Protect(sensitiveData);
11.        // Setting encrypted cookie
12.        context.Response.Cookies.Append
13.        ("MyProtectedCookie", encryptedData);
14.        await context.Response.WriteAsync
```

```

14.         (@"Cookie has been set.
15.             Refresh the page to see the decrypted data.");
16.     }
17.     else
18.     {
19.         // Retrieving and decrypting the data
20.         var decryptedData =
protector.Unprotect(encryptedCookie);
21.         await context.Response.WriteAsync
22.             ($"Decrypted Cookie Data: {decryptedData}");
23.     }
24. });

```

A **MyProtectedCookie** encrypted cookie is set in this API if it does not already exist. In subsequent requests, the same cookie is decrypted, and the content is returned. By using this type of data protection, we will ensure the protection and confidentiality of the data stored directly on the user's computer but be careful, any data stored on the client is potentially unsafe. We have to be very careful about what we leave on the client and how we leave it.

Transport Layer Security

Another way to ensure that all data transmitted through the network are encrypted is to use the technique called **Transport Layer Safety (TLS)**. To do this, we must always use HTTPS to allow the connection to our server. This helps us to prevent unauthorized access and data breaches during the transmission itself by safeguarding sensitive data while maintaining the integrity between our servers and clients. Proper implementation of TLS is crucial to the security of our web communications and to protecting user data from interception and malicious manipulation. Enable the HTTPS as follows:

```

1. builder.Services.AddHttpsRedirection(options =>
2. {
3.     options.HttpsPort = 443;
4. });
5. var app = builder.Build();
6. app.UseHttpsRedirection();
7. app.Run();

```

Adopting TLS and enforcing HTTPS are essential practices for securing our network communications, ensuring data privacy and protecting against potential threats.

Logging and monitoring

Logging and monitoring are critical for maintaining security in ASP.NET Core applications. Implement comprehensive logging to capture detailed information about application activities including user actions, errors and system events. Use ASP.NET Core's built-in logging framework or integrate with third-party services like Serilog, ELK Stack or Azure monitor for robust logging capabilities. The constant monitoring of logs (automatic or manual, even if not recommended) to detect suspicious activities or potential security breaches allow us to notice them in real time. By implementing warning mechanisms, you can quickly detect anomalies, critical problems or unexpected security flaws. Effective logging and monitoring improve the perception of application behavior, facilitate rapid response to incidents and significantly contribute to overall application security.

Implementing Serilog logging

Logging is one of the better instruments to identifying and solving security issues, as it provides detailed records of system activities. Logs help to monitor suspicious behavior, track breaches and analyze possible attacks.

Serilog

Serilog is definitely a great choice for structured event logging in ASP.NET Core applications. This library simplifies all logging management by allowing us to capture detailed data with a rich structure. Serilog supports writing logs in several different environments and destinations. The flexibility of the library and its configurability make it the ideal tool for application monitoring and troubleshooting. The reference website for Serilog has an exhaustive documentation able to guide each developer to integrate it properly.

In .NET 8, Serilog was added to the ASP.NET Core templates, but in .NET 9, it was removed. This decision by Microsoft was not due to a lack of validity of the product but rather to keep the templates as

neutral as possible regarding logging providers, allowing developers the freedom to choose the solution that best suits their needs. As we just mentioned, Serilog remains an excellent product and can be added to our projects at any time by simply configuring it as the logging provider in the application's startup.

Sensitive data

Whenever we write a line in our logs, always remember that we must never record sensitive data such as passwords, credit card numbers and other personal information both to protect the privacy of users and to comply with security regulations. The recording of such data may expose users to unauthorized access and potential breaches. Ensuring that none of the above-mentioned sensitive information is recorded helps reduce the risk of identity theft and financial fraud to our users and customers.

Monitoring

Constantly monitoring our application is essential for quickly spotting and addressing security threats. By regularly analysing network traffic, system activities and user behaviour, we can identify unusual patterns and potential breaches as they occur. This proactive approach allows our application to respond immediately, minimizing the impact of any threats. Implementing continuous monitoring helps keep our security robust, ensures that vulnerabilities are promptly dealt with and maintains the overall integrity and security of our systems. It is like having a vigilant guard, always on duty to protect our digital environment.

Use middleware

Custom middleware inspects and logs HTTP request and response details for better visibility. As we have seen in [Chapter 6, Middleware and Extensibility](#), we have to build the middleware class as follows:

1. `public class RequestLoggingMiddleware`
2. `{`
3. `private readonly RequestDelegate next;`
4. `private readonly ILogger<RequestLoggingMiddleware>`
`logger;`
5. `public RequestLoggingMiddleware`

```

6.     (RequestDelegate next
7.     , ILogger< RequestLoggingMiddleware > logger)
8.     {
9.         this.next = next;
10.        this.logger = logger;
11.    }
12.    public async Task Invoke(HttpContext context)
13.    {
14.        var request = context.Request;
15.        logger.LogInformation
16.            ("Request: {Method} {Path}"
17.            , request.Method, request.Path);
18.        await next(context);
19.        logger.LogInformation
20.            ("Response: {StatusCode}"
21.            , context.Response.StatusCode);
22.    }
23. }

```

Now, we have to register it in the **Program.cs** dependency injection as follows:

```
1. app.UseMiddleware< RequestLoggingMiddleware >();
```

In this way, we will have a complete log of all the activities combining middleware technique with Serilog logging.

Application Insights

Another way to keep track of everything happening in our application is Azure Application Insights. It monitors application performance by tracking request rates, response times and dependencies. This helps identify issues, optimize user experiences and ensure the application runs smoothly, providing valuable insights for maintaining and improving overall performance. To use it we several options. One of the options is to install the **nuget** package

Microsoft.ApplicationInsights.AspNetCore

Then, configure the service as follows:

```
1. builder.Services.AddApplicationInsightsTelemetry
```

2. (`"ApplicationInsights:InstrumentationKey"`);

Another approach is to directly link the application or service to an Application Insights resource through the Azure portal. This automatically adds the instrumentation key to the environment variables, eliminating the need to modify the application to integrate the Application Insights package. The insights will be available on Azure portal or downloading them into our application even if is better have a separate application to retrieve the insights.

Authorization management

Authorization and authentication in ASP.NET Core ensure secure access to resources. Authentication verifies user identity, typically using services like identity or external providers (e.g., OAuth). Authorization determines user permissions via policies, roles or claims. These mechanisms work together to protect applications and manage user access effectively.

Implementing authentication

Before managing the application authorization, we have to know who is connected to our system or more simply who is going to use the application, to properly apply some rules that fit the user. A proper authentication mechanism is vital to ensure that only authorized users can access to the application. There are two main ways as following:

- **ASP.NET Core Identity:** Use ASP.NET Core Identity for user management and authentication.
- **External providers:** Integrate external authentication providers like Google, Facebook, or Microsoft for additional security.

For this part and only for an example, following is the Google authentication, so the first step is to install the Google authentication **nuget** package, **Microsoft.AspNetCore.Authentication.Google**, then add the following code:

1. `builder.Services.AddAuthentication()`
2. `.AddGoogle(options = >`

```

3. {
4.     options.ClientId = builder
5.
6.     .Configuration["Authentication:Google.ClientId"];
7.     options.ClientSecret = builder
8.         .Configuration["Authentication:Google.
9.             ClientSecret"];
10. });

```

At this point, our user is known in the system. We can apply a proper authorization to manage the application contents.

Implementing authorization

Authorization ensures that authenticated users have access only to resources they are permitted to use. There are two main ways to perform authorization in an application as follows:

- **Role-based authorization:** Use role-based authorization to control access to different parts of the application based on user roles.
- **Policy-based authorization:** Define policies for more granular control over access permissions. Simply in our application we could define something like the following:

```

1. builder.Services.AddAuthorization(options =>
2. {
3.     options.AddPolicy("AdminOnly"
4.         , policy => policy.RequireRole("Admin"));
5. });

```

Using JSON Web Token

One of the best practices to implement the authentication in a web application is the stateless authentication.

Stateless authentication is a method where the server does not maintain user session state. Instead, all necessary information for authentication and authorization is contained within a token like a **JSON Web Token (JWT)**, that the user sends with each request. The token includes metadata, user information and a signature.

When the server receives a request, it verifies the token's signature and decodes the user info without needing to check a database. This method enhances scalability and reduces server load since each request is independent but it also presents challenges like token revocation and security management.

Two best practices to better implement this technique are as follows:

- **Token expiration:** Set an appropriate expiration time for tokens to minimize the impact of a compromised token.
- **Token storage:** Store tokens securely, preferably in HTTP-only cookies to mitigate XSS attacks.

Stateless authentication enhances scalability by treating each request independently, using tokens for user info but requires careful token security management.

Usage of HTTPS and SSL/TLS certificates

The basis of security of every web server today is given by HTTPS. It is a protocol that ensures secure client-server communication. HTTPS, is an extension of the standard HTTP protocol to which SSL/TLS protocols have been added to provide an encrypted connection between a web server and a client. This encryption is the basis for the protection, integrity and confidentiality of the data exchanged. There are three main reasons why HTTPS is essential for web server security, as follows:

- **Data encryption:** HTTPS uses, as we said, SSL/TLS protocols to encrypt data transmitted between the web server and the client. This means that all information, such as login credentials, personal data and financial transactions are protected from interception and therefore also from unauthorized access. Encryption turns data into a secure and unreadable format. It can only be decoded and interpreted by the recipient, ensuring that sensitive information are always safe from listeners and cybercriminals.
- **Data integrity:** The SSL/TLS protocols used by HTTPS in addition to ensuring the confidentiality of data and their integrity. Data integrity means that the information

exchanged between the server and the client cannot be altered during transit without the recipient being aware of it. If a manipulation occurs, the connection is immediately disconnected, warning both sides of the potential threat. This ensures that the data received from the client is exactly what the server sent, free of unlawful modifications or alterations.

- **Compliance with regulations:** Many international security standards and regulations explicitly require the use of HTTPS to protect user data. These regulations are designed to ensure that organizations treat user information responsibly and securely. The implementation of HTTPS helps organizations comply with these regulations, avoiding potential penalties and reputation damage that may arise from data breaches or non-compliance with current regulations.

HTTPS is essential for web server security because it provides robust data encryption, ensures data integrity and ensures compliance with regulations. These combined advantages make HTTPS a key component of any online communications security Strategy and ensure the integrity and confidentiality of user information. Without HTTPS, web servers and their users would be vulnerable to a wide range of security threats.

Protecting against common attacks

There are some well-known attacks where we as developers we must protect our application. Let us see the most diffused and common attacks.

Cross-site scripting

Cross-site scripting (XSS) attacks occur when an attacker injects malicious scripts into web content that is rendered on the client-side. These scripts can steal cookies, session tokens or other sensitive information, manipulate website content or redirect users to malicious sites, compromising the security and integrity of user interactions. To prevent this attack there are two main directives to take in mind as follows:

- **Input validation:** Always validate and sanitize user inputs.

- **Output encoding:** Encode outputs to prevent execution of malicious scripts.

In ASP.NET MVC there is an `HtmlHelper` to ensure the encoding of user input able to prevent XSS attacks as follows:

1. `@Html.Encode(userInput)`

In ASP.NET Core, all user inputs are automatically encoded by design, so it is no longer necessary to use this syntax.

SQL injection

SQL injection attacks is one of the most classic that involve injecting malicious SQL queries through user inputs, exploiting vulnerabilities in an application's software to manipulate the database. These attacks can result in unauthorized access to sensitive data, data loss or corruption. They can also allow attackers to execute administrative operations, compromising the application's security and integrity. There are some rules and recommendations to avoid this kind of attack as follows:

- **Parameterized queries:** Always use parameterized queries to prevent SQL injection. Let us see the following code:

```
1. using (var command = new SqlCommand
2.     ("SELECT * FROM Users
3.     WHERE Username = @Username"
4.     , connection))
5. {
6.     command.Parameters.AddWithValue("@Username",
7.     username);
8. }
9.
```

This is the right way to prevent this kind of attack.

- **Query string:** Another important tip is to avoid including SQL queries in the URL query strings. Although this may seem obvious today, it is worth pointing out because a decade ago it was a common practice and an easy target for attacks. Recently I have again encountered a similar practice that exploits query string to pass potentially manipulable

data. Obviously from the point of view of a developer, it could be the fastest way to pass data but even the worst. Integrating SQL into query strings exposes the database to SQL injection attacks, in which malicious users can manipulate queries to obtain unauthorized access, retrieve sensitive data, or corrupt the database.

Modern web development practices support the use of prepared statements or parameterized queries to protect against these vulnerabilities, ensuring the security and integrity of the database and preventing the direct execution of SQL via user input.

Points to remember

- **C-I-A:** In information security, C-I-A means confidentiality, integrity, and availability
- **The three Au's:** Authentication, authorization, audit
- **Authentication:** Verifies who the user is
- **Authorization:** Determines what the user can do
- **Auditing:** Tracks what the user has done

Conclusion

Securing ASP.NET Core application is an ongoing process that requires vigilance and adherence to best practices. By following the guidelines outlined in this chapter, we can significantly enhance the security of your backend, protect sensitive data and ensure that your application is resilient against common threats.

In the next chapter, we will explore advanced performance optimization techniques for ASP.NET Core applications.

Multiple choice questions

1. Which of the following is a best practice for both backend and frontend security in ASP.NET Core?
 - a. Use the same password for all accounts
 - b. Disable security headers

- c. Regularly update and patch all software and libraries
 - d. Store sensitive information in local storage
- 2. What is the main purpose of authorization management in ASP.NET Core?**
- a. To manage database schemas
 - b. To control user access to resources
 - c. To optimize web performance
 - d. To handle email notifications
- 3. Why is HTTPS important for securing web applications?**
- a. It speeds up the website
 - b. It encrypts data between the client and server
 - c. It allows for unlimited data storage
 - d. It provides free hosting
- 4. What is the benefit of using SSL/TLS certificates on a website?**
- a. They provide free hosting
 - b. They authenticate the server to the client and encrypt data
 - c. They increase website traffic
 - d. They disable security features
- 5. How does HTTPS help ensure the integrity of data transmitted between a client and a server?**
- a. By compressing the data
 - b. By encrypting the data and detecting tampering during transmission
 - c. By backing up data regularly
 - d. By increasing data storage

Answer key

Key terms

- **Authorization management:** The process of controlling user access to resources based on their permissions.
- **Authentication management:** The process of controlling user access by verifying identities, ensuring secure logins and managing credentials.
- **HyperText Transfer Protocol Secure (HTTPS):** A protocol for secure communication over a computer network, encrypting data exchanged between the client and server.
- **Anti-forgery token:** A security token used to prevent **Cross-Site Request Forgery (CSRF)** attacks by ensuring that form submissions are legitimate.
- **Data encryption:** The process of converting data into a secure format that can only be read by someone with the correct decryption key.
- **SQL injection:** A code injection technique that exploits vulnerabilities in an application's software to execute malicious SQL queries.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 14

Securing Web Applications Effectively

Introduction

In this chapter, we will explore essential aspects of ASP.NET Core application protection. We will start by examining authentication and authorization, which verify user identities and control resource access. Through a basic implementation of JWT token-based authentication and authorization, we will discover how robust systems can ensure that only authorized users access specific features. Next, an overview on data protection and encryption to safeguard sensitive information. We also examine common web vulnerabilities such as XSS and CSRF, and how to avoid them by implementing adequate security measures.

Structure

In this chapter, we will cover the following topics:

- Authentication and authorization
- Data protection and encryption
- Cross-site scripting and cross-site request forgery protection

Objectives

By the end of this chapter, readers will be able to understand and implement authentication and authorization in ASP.NET Core, configuring services and mechanisms such as cookies, JWT and third-party providers. We will learn how to manage user roles and policies for control access to resources. We will understand the importance of data protection and encryption, implementing techniques to protect sensitive data. We will also see how to protect our application from XSS and CSRF. Finally, we will learn how to implement effective logging and monitoring to control and detect potential security threats and issues in a timely manner.

Authentication and authorization

In *Chapter 13, Security in ASP.NET Core*, we have seen the main principles of authentication and authorization. Here, we will see how to effectively implement the authentication in a browser and store information of this authentication in a JWT token generated by server. If the token is generated by the server properly on the client, we will store it and decorate each request with that token. Let us see in detail how to do that.

Authentication using JWT token

As we have seen in *Chapter 13, Security in ASP.NET Core*, when we use **JSON Web Token (JWT)** token the server will be stateless. In practical terms if we implement authentication and authorization based on JWT token our server (backend) should not store which user is logged. When a request arrives, it will be correlated with the JWT Token and only the server must be able to interpreter. In case of authorization, the client should ask to server what his role is or what level of privileges the client has. It will be the server that will answer the client properly. This mechanism will guarantee that on client will not have any logic able to interpret the token so that

nobody should grab that kind of information.

JWT token

JWT is a compact and autonomous way to transmit information between two parties via JSON object. This information can be reliable because it is digitally signed and can be easily verified. A JWT consists of three main parts, that is, **header**, **payload**, and **signatures**, separated by points ('.'). In detail the three parts are composed as follows:

- **Header:** The header usually contains two fields: the token type, which is always JWT, and the signing algorithm used, such as HMAC SHA256 or RSA. This JSON is then encoded in Base64Url.
- **Payload:** The payload contains the claims, which are statements about an entity (usually, the user) and additional data such as roles and authorizations. There are three types of claims: registered, public and private. *Registered* claims are a series of default claims recommended to provide a set of useful information, such as **emitter (iss)**, **expiry time (exp)**, **object (sub)** and **audience (aud)**. Public claims can be multiple and defined by those who emit and use the JWT, but to avoid name collisions, they should be registered. *Private* claims are custom claims created to share information between parties that agree on their use. The payload is also encoded in Base64Url.
- **Signature:** The signature is created by taking the encoded header, the encrypted payload, a secret key (known only to the server) and the algorithm specified in the header. So, if everything matches the JWT will be digitally signed.

A full JWT looks as follows:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJmM0NTY3ODkwIiwiaWF0Ij01pvaG4gRG9lIiwiaWQiOiJYWWRtaW4iRydWV9.SflKxwRJSMeKKF2QT4fwpMeJf36POl

In this example, there are three visible parts separated by points.

Use of JWT

JWT is commonly used for both authentication and authorization.

In systems using JWT tokens, as we said before, the server is stateless as the information is read from the JWT token. For example, when a user logs in, the server generates a unique JWT token, which must be included in every request. This allows the server to authenticate requests without querying the database or maintaining user sessions. Additionally, a JWT can verify if a user has the necessary permissions to access a resource. The primary advantage of using a JWT is its autonomy, containing all necessary information and reducing database access for permissions checks. Being signed and encrypted, it ensures the token's integrity and confidentiality. As an open standard (RFC 7519), it is supported by many programming languages.

Using JWT in ASP.NET Core authentication

In ASP.NET Core, JWTs are commonly used to handle user authentication and authorization in a secure and stateless manner.

To ensure the JWT token will be managed properly, first we have to install the **Microsoft.AspNetCore.Authentication.JwtBearer** NuGet package.

Then, in **Program.cs** file, we will add the configuration of the JWT authentication service as follows:

```
1. builder.Services.AddAuthentication(options =>
2. {
3.     options.DefaultAuthenticateScheme
4.         = JwtBearerDefaults.AuthenticationScheme;
5.     options.DefaultChallengeScheme
6.         = JwtBearerDefaults.AuthenticationScheme;
7. })
8. .AddJwtBearer(options =>
9. {
10.     options.TokenValidationParameters = new
    TokenValidationParameters
11.     {
12.         ValidateIssuer = true,
13.         ValidateAudience = true,
14.         ValidateLifetime = true,
```

```

15.     ValidateIssuerSigningKey = true,
16.     ValidIssuer = "Issuer",
17.     ValidAudience = "Audience",
18.     IssuerSigningKey = new SymmetricSecurityKey
19.         (Encoding.UTF8.GetBytes("Key"))
20. };
21. });
22. builder.Services.AddControllers();

```

After the configuration, ensure the authentication middleware is added in the **Program.cs** as follows:

```

1. app.UseRouting();
2. app.UseAuthentication();
3. app.UseAuthorization();
4. app.MapRazorPages();

```

Now, we are ready to generate a proper JWT token. We will do that in a separate class named **JwtManagement.cs** where we will put all the methods and manipulations for our JWT token. In that class we will implement a method to generate JWT tokens upon successful user authentication, as follows:

```

1. internal string GenerateJwtToken(User user)
2. {
3.     var securityKey = new SymmetricSecurityKey
4.         (Encoding.UTF8.GetBytes("Key"));
5.     var credentials = new SigningCredentials
6.         (securityKey, SecurityAlgorithms.HmacSha256);
7.     var claims = new[]
8.     {
9.         new Claim(JwtRegisteredClaimNames.Sub
10.             , user.Username),
11.         new Claim(JwtRegisteredClaimNames.Jti
12.             , Guid.NewGuid().ToString())
13.     };
14.     var token = new JwtSecurityToken("Issuer", "Audience",
15.         claims,
16.         expires: DateTime.Now.AddMinutes(30),

```

```
16.     signingCredentials: credentials);
17.     return new JwtSecurityTokenHandler().WriteToken(token);
18. }
```

Under the claims array, we can add some other personal claims as role or particular management of the user. For instance, we could decide to add some roles to our user, then we could add a claim similar as following:

```
new Claim(ClaimTypes.Role, "ADMIN"),
new Claim(ClaimTypes.Role, "CONFIGURATOR"),
new Claim(ClaimTypes.Role, "BRANCH")
```

In this case, our user will have the role of an administrator that can configure the application only in the branch office of the company.

Securing WebAPI

In our backend, we have all the APIs able to access the system. In this case, we have to secure the APIs before the client access it. For an example, we can imagine three methods: the first for *Administrators*, the second for the role of *User* and the third for everyone.

To achieve our goal to securize each endpoint of our controller, we can use some attributes. First we can decorate each method with a `[Authorize(Roles = "RoleName")]` to be sure that the method is consumed only for the authorized role clients. Our controller will be transformed as follows:

```
1. [ApiController]
2. [Route("api/[controller]")]
3. public class ExampleController : ControllerBase
4. {
5.     [HttpGet("admin")]
6.     [Authorize(Roles = "Admin")]
7.     public IActionResult GetAdminData() => Ok();
8.     [HttpGet("user")]
9.     [Authorize]
10.    public IActionResult GetUserData() => Ok();
```

```

11. [HttpGet("public")]
12. [AllowAnonymous]
13. public IActionResult GetPublicData() => Ok();
14. }

```

As we can see, we have explicit authorization attributes able to **filter** the authenticated user based on their specific role. As we have seen in section *Using JWT in ASP.NET Core authentication*, we configured a middleware that can control each JWT token passed to our endpoints so that it can control properly the access. This middleware is directly implicated to the process and if the role is not in phase on the ones previewed, the middleware quit the request with a **401 Unauthorized** answer to client.

Client side usage

After the server generates JWT, the client (e.g., a browser or mobile app) stores it, typically in local storage or protected cookies. This token will be passed to each request to server but when we retrieve the role and information from server, we have to display them into a proper view. In ASP.NET Core applications we have a helper tag `<authorize>` in ASP.NET Core and `<AuthorizeView>` in Blazor that limits the access to the included code according to the user role, as follows:

```

1. <AuthorizeView Roles="Admin">
2.   <Authorized>
3.     <p> The Administrator is authorized to see contents </p>
4.   </Authorized>
5.   <NotAuthorized>
6.     <p> The user is
7.       <strong> NOT </strong>
8.       authorized to see this content.
9.     </p>
10.   </NotAuthorized>
11. </AuthorizeView>

```

In [Chapter 18, Advanced User Interfaces with Blazor](#), we will see in depth on how to work with authorizations in UI made with Blazor.

Data protection and encryption

Data protection involves practices and technologies designed to ensure that sensitive data is accessible only to authorized individuals and is protected against unauthorized access, manipulation, or loss. Encryption is a primary method used for data protection. Encryption transforms readable data (plaintext) into an encoded format (ciphertext) that can only be read by those possessing the appropriate decryption key.

Principles and rules of encryption

To protect data effectively, it is essential to follow fundamental encryption principles. These principles ensure data remains secure, accessible only to authorized individuals, and free from unauthorized alterations. Implementing these key principles is crucial for maintaining the confidentiality, integrity, authenticity, and non-repudiation of sensitive information. By adhering to these guidelines, organizations can build robust security frameworks, safeguarding data and maintaining user trust. There are four main principles to consider when we encrypt data, as follows:

- **Confidentiality:** Only authorized individuals can access protected data. This is achieved by converting plaintext into ciphertext, which can only be decrypted by someone with the correct key, preventing unauthorized access.
- **Integrity:** Ensures data remains unchanged during transit or storage. Any tampering or alteration is detected, maintaining trust in the data's accuracy and reliability.
- **Authenticity:** Verifies the identity of users or systems accessing the data, ensuring it comes from a trusted source. Authentication mechanisms prevent impersonation and unauthorized access.
- **Non-repudiation:** Ensures users cannot deny actions performed on the data. Digital signatures and logging provide evidence of actions, crucial for legal and forensic purposes.

Following these encryption principles is essential for creating a secure and trustworthy data protection strategy.

Encryption in ASP.NET Core

In ASP.NET Core, data protection and encryption can be managed using the built-in **data protection** system and various encryption libraries such as **System.Security.Cryptography**.

Configuring and using data protection

To secure sensitive data in our ASP.NET Core application, we have to ensure that the nuget package **Microsoft.AspNetCore.DataProtection** is referenced in the project. Now, we can configure the data protection service in the **Program.cs** file by adding it to the service collection, specifying the key storage location, and setting the application name to ensure data isolation across different applications as follows:

1. `builder.Services.AddDataProtection()`
2. `.PersistKeysToFileSystem(new DirectoryInfo(@"C:\keys"))`
3. `.SetApplicationName("MyApp");`

Following is how the code works and why:

- **builder.Services.AddDataProtection()**. This line adds data protection services to the application's service container. Data protection is used to protect sensitive information such as authentication cookies, CSRF tokens and other data that needs to be encrypted or signed.
- **PersistKeysToFileSystem(new DirectoryInfo(@"C:\keys"))** this line configures the data protection system to use the file system as a key archive. The keys are stored in the specified folder (in this case, C:\key). These keys are used to encrypt and decrypt protected data. It is important to adequately protect this folder to ensure the security of the keys.
- **SetApplicationName("MyApp")** this line defines the application name for the data protection system. The application name is used to isolate protected data between different applications. If two applications share the same key archive but have different application names, they will not be able to decipher each other's data.

In summary, the three lines of code added will configure the data protection system of an ASP.NET Core application, specifying a

physical path where to store the encryption keys and setting a name for the application to isolate the protected data.

Use the data protection system

After configuring the protection system, we have to use it. The best fit is use it in a controller to protect and unprotect data. Following is a simple controller:

```
1. public class HomeController(IDataProtectionProvider provider)
2.     : Controller
3. {
4.     private readonly IDataProtector protector
5.         = provider.CreateProtector("SamplePurpose");
6.     public IActionResult Index()
7.     {
8.         string protectedData = protector.Protect("Sensitive data");
9.         string unprotectedData =
10.             protector.Unprotect(protectedData);
11.         ViewData["ProtectedData"] = protectedData;
12.         ViewData["UnprotectedData"] = unprotectedData;
13.         return View();
14. }
```

In this controller, we will have a simple **Protect** and **Unprotect** methods able to protect and unprotect our sensible data. For simplicity, our controller will return a view that stores in **ViewData** both.

Implementing encryption with AES

The implementation of **Advanced Encryption Standard (AES)** encryption in ASP.NET Core is a process that ensures the security of sensitive data. AES is a symmetrical encryption algorithm widely recognized for its efficiency and security. In ASP.NET Core, we can use the **System.Security.Cryptography** namespace to access AES features. To begin, create an **AesCryptoServiceProvider** or **AesManaged** instance, configuring the key and **initialization vector (IV)** needed for encryption and decryption. Next, we can use a **CryptoStream** object to encrypt or decrypt the flow data. This approach allows us to protect critical information, such as

passwords or financial data, ensuring that only authorized parties can access the data clearly.

Following is a simple example:

```
1. public static string EncryptString
2.     (string plainText, byte[] key, byte[] iv)
3. {
4.     using (var aes = Aes.Create())
5.     {
6.         aes.Key = key;
7.         aes.IV = iv;
8.         using (var encryptor = aes.CreateEncryptor
9.             (aes.Key, aes.IV))
10.            using (var ms = new MemoryStream())
11.            {
12.                using (var cs = new CryptoStream
13.                    (ms, encryptor, CryptoStreamMode.Write))
14.                using (var sw = new StreamWriter(cs))
15.                {
16.                    sw.Write(plainText);
17.                }
18.                return Convert.ToBase64String(ms.ToArray());
19.            }
20.        }
21.    }
22. public static string DecryptString
23.     (string cipherText, byte[] key, byte[] iv)
24. {
25.     using (Aes aes = Aes.Create())
26.     {
27.         aes.Key = key;
28.         aes.IV = iv;
29.         using (var decryptor = aes.CreateDecryptor
30.             (aes.Key, aes.IV))
31.         using (var ms = new MemoryStream
32.             (Convert.FromBase64String(cipherText)))
33.         using (var cs = new CryptoStream
34.             (ms, decryptor, CryptoStreamMode.Read))
```



```

35.     using (var sr = new StreamReader(cs))
36.     {
37.         return sr.ReadToEnd();
38.     }
39. }
40. }

```

The use of this class will be as follows:

```

1. public class AesEncryptionImplementation()
2. {
3.     public AesEncryptionImplementation()
4.     {
5.         string original = "Here is some data to encrypt!";
6.         using (Aes aes = Aes.Create())
7.         {
8.             string encrypted = AesEncryption.EncryptString
9.                 (original, aes.Key, aes.IV);
10.            Console.WriteLine($"Encrypted: {encrypted}");
11.            string decrypted = AesEncryption.DecryptString
12.                (encrypted, aes.Key, aes.IV);
13.            Console.WriteLine($"Decrypted: {decrypted}");
14.        }
15.    }
16. }

```

With these implementations, we have configured both data protection and encryption in an ASP.NET Core application.

Cross-site scripting and cross-site request forgery protection

It is widely accepted that all software has bugs, though exceptions exist for trivial, provably correct, or highly engineered software in critical systems. For most software, recognizing the ubiquity of bugs is essential for secure coding, as some bugs can be exploited by attackers. Vulnerabilities are bugs that can cause harm, and distinguishing them from harmless bugs can be challenging. For instance, a harmless bug might be a web page layout issue that does not affect functionality, whereas a harmful bug could expose an administrative interface to the internet, posing a severe security

threat. The spectrum of bugs includes many that require subjective judgment to assess their potential harm. It is advisable to err on the side of caution and address bugs that might be vulnerabilities. Most projects have databases of known bugs, or technical debts, yet rarely aim to eliminate them all. Fixing bugs is generally easier than proving they are harmless, and prioritizing security bugs is crucial. Recognizing all software has bugs is crucial, as some may lead to vulnerabilities like XSS and CSRF.

Cross-site scripting vulnerability

XSS is a security vulnerability found in web applications. It allows attackers to inject malicious scripts into content that is then delivered to other users. For example, we can consider a blog that allows comments from other users. The malicious script will be sent to the server under the form of comment and each user that lands on that page is infected by that script. These scripts can be used to steal cookies, session tokens, or other sensitive information, manipulate the content of the web page, or perform other malicious activities on behalf of the user.

Following is an ASP.NET Core example that demonstrates how an XSS vulnerability can occur. For simplicity this example includes a form that takes user input and displays it.

As usual we define a controller that build a view, as follows:

```
1. public class HomeController : Controller
2. {
3.     [HttpGet]
4.     public IActionResult Index() => View();
5.     [HttpPost]
6.     public IActionResult SubmitForm(string userInput)
7.     {
8.         ViewData["Message"] = $"You entered: {userInput}";
9.         return View("Index");
10.    }
11. }
```

Then, the view should be as follows:

```
1. @{
2.     ViewData["Title"] = "Home Page";
```

```

3. }
4. <h1 class="display-4">Welcome</h1>
5. <form method="post" asp-action="SubmitForm">
6.   <div class="form-group">
7.     <label for="userInput">Enter something:</label>
8.     <input type="text"
9.       class="form-control"
10.      id="userInput" name="userInput">
11.   </div>
12.   <button type="submit"
13.     class="btn btn-primary">
14.     Submit
15.   </button>
16. </form>
17. <div>
18.   @Html.Raw(ViewData["Message"])
19. </div>

```

An attacker could enter the following script in the input field:

```
<script>alert('XSS Attack');</script>
```

When this input is submitted, it is rendered directly on the page, causing the script to execute and showing an alert box. In this case, there is only an alert box, but imagine if a more complex script will be injected, saved on the database on server and executed on each client that in future shows the commented page. For instance, a script that read cookies or local storage and send the information to a remote server in a totally transparency from the client.

Mitigating XSS vulnerability

To mitigate XSS vulnerabilities, we should always sanitize user input before displaying it. ASP.NET Core provides utilities to help with this.

In the controller we need to add the following line:

```
var sanitizedInput =
System.Net.WebUtility.HtmlEncode(userInput);
```

Then, this **sanitizedInput** will be the data passed to the view as follows:

```
ViewData["Message"] = $"You entered: {sanitizedInput}";
```

In this way, the framework provides sanitization of user input to prevent the XSS attack.

Cross-site request forgery vulnerability

CSRF is a type of attack that tricks a user into performing actions they did not intend to, within a web application in which they are authenticated. For example, by visiting a malicious website, the attacker can cause the user's browser to make requests to a trusted website where the user is logged in, potentially changing data or performing unintended actions.

Following is an ASP.NET Core example that demonstrates how a CSRF vulnerability can occur. This example includes a form that performs an action without CSRF protection.

Consider the same vulnerable controller seen in previous section and the same view.

The attack is roughly similar to an XSS attack but in a more profound way. In CSRF, a complete form is injected as user input, shown to the user, and executed using the user's credentials. For an example, the following HTML should be injected:

```
1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4.   <meta charset="UTF-8">
5.   <title>CSRF Attack</title>
6. </head>
7. <body>
8.   <form action="https://malicious-website.com/Home/SubmitForm" method="post">
9.     <input type="hidden" name="userInput" value="malicious input">
10.    <input type="submit" value="Submit request">
11.  </form>
12.  <script>
13.    document.forms[0].submit();
14.  </script>
15. </body>
```

16. `</html>`

When a logged-in user visits this page, the form will be submitted to the malicious website without their consent, potentially causing unwanted actions.

Mitigating CSRF vulnerability

To mitigate CSRF vulnerabilities, the framework helps us providing an attribute we should include as anti-forgery tokens and validate them on the server. The controller method should be as follows:

```
1. [HttpPost]
2. [ValidateAntiForgeryToken] // Protect against CSRF
3. public IActionResult SubmitForm(string userInput)
4. {
5.     ViewData["Message"] = $"You entered: {userInput}";
6.     return View("Index");
7. }
```

In the view (`cshtml` file), we can use `@Html.AntiForgeryToken()` attribute in our form, that generates a hidden field with a token unique to the user's session, which the server validates upon submission. The `[ValidateAntiForgeryToken]` attribute on the `SubmitForm` action method ensures the server checks this token. If the token is missing or invalid, the server rejects the request. These changes prevent malicious form submissions from another site because the anti-forgery token would not match, effectively mitigating the CSRF vulnerability. This approach ensures that only legitimate requests are processed, enhancing security.

Conclusion

In this chapter, we provided a comprehensive overview of essential security concepts in ASP.NET Core. We explored the mechanisms of authentication and authorization, ensuring that only legitimate users can access and perform actions within the application. We also delved into data protection and encryption, highlighting the importance of safeguarding sensitive information. Additionally, we addressed XSS and CSRF protection, emphasizing strategies to prevent these common web vulnerabilities.

In the next chapter, we will see how to treat errors in ASP.NET Core

application and how to properly log them.

Points to remember

- **Authentication and authorization:** Ensure only legitimate users can access and perform actions.
- **Data protection and encryption:** Safeguard sensitive information using robust encryption mechanisms.
- **XSS protection:** Sanitize user input to prevent malicious scripts execution.
- **CSRF protection:** Use anti-forgery tokens to ensure requests are legitimate.

Multiple choice questions

1. What is the main purpose of authentication in ASP.NET Core?
 - a. Encrypt data
 - b. Sanitize user input
 - c. Ensure only legitimate users can access the application
 - d. Generate anti-forgery tokens
2. What is Cross-Site Scripting (XSS)?
 - a. A way to encrypt user data
 - b. An attack where malicious scripts are injected into webpages
 - c. A method to verify user identity
 - d. A technique to protect against CSRF
3. Which of the following is a measure to prevent CSRF attacks?
 - a. Encrypting user passwords
 - b. Using anti-forgery tokens
 - c. Sanitizing user inputs
 - d. Disabling JavaScript
4. What does data encryption help to achieve?

- a. Speed up data processing
- b. Make data unreadable to unauthorized users
- c. Validate user input
- d. Prevent SQL injection attacks

5. What is the main purpose of authorization in ASP.NET Core?

- a. Encrypt user data
- b. Verify user identity
- c. Determine what actions a user can perform
- d. Generate anti-forgery tokens

Answer key

Key terms

- **Authentication:** Verifying the identity of a user or system.
- **Authorization:** Determining what actions an authenticated user can perform.
- **Encryption:** Converting data into a secure format that is unreadable without a decryption key.
- **XSS:** A vulnerability that allows attackers to inject malicious scripts into webpages.
- **CSRF:** An attack that tricks a user into performing actions they did not intend to within a web application.
- **Anti-forgery token:** A unique token used to protect against CSRF attacks.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 15

Error Handling

Introduction

In modern web development, maintaining the stability and performance of applications is very important. This chapter explores essential techniques and tools in ASP.NET Core that help developers build robust and reliable systems. We begin by examining how to capture and analyze application data, an essential aspect of understanding an application's behavior and diagnosing issues. We then explore methods for managing unexpected issues, ensuring that applications handle problems gracefully without impacting the user experience. This includes setting up mechanisms to catch and manage errors in a controlled manner, preventing disruptions and maintaining a smooth operation. Additionally, we focus on techniques for monitoring and evaluating application health. This involves leveraging tools that provide real-time insights into system performance and behavior, enabling proactive management and optimization.

Structure

In this chapter, we will cover the following topics:

- Logging and error tracking
- Exception handling
- Monitoring and diagnostic tools

Objectives

By the end of this chapter, readers will be able to handle exceptions gracefully to maintain application stability and utilize various monitoring and diagnostic tools to assess and improve system performance. The chapter aims to provide practical knowledge on capturing and analysing application data, managing errors and exceptions without disrupting user experience, and proactively monitoring the health and behavior of applications to ensure reliability and efficiency. By the end of this chapter, we will be equipped with practical knowledge and strategies to enhance the reliability and efficiency of your ASP.NET Core applications, ensuring they run smoothly in any environment.

Logging and error tracking

There is no application bug-free. We can predict each behavior, create a lot of use cases or edge cases but there will be always something of unexpected. In that case we cannot permit that our application falls in a crash. In any case, we have to catch the unexpected behavior and inform the user that something went wrong. In the meantime, we have to save all of the possible information and steps to reproduce the bug so that we, as developers, can fix properly the issue but we cannot log everything. As developers, we cannot include everything in the logs, so we need to make informed decisions.

Following are some initial guidelines to achieve balanced logging:

- **Clarity and context:** Log clear and contextual information, including timestamps, involved modules, and details useful for diagnosis.
- **Log levels:** Classify messages by importance (Debug, Info, Warning, Error, Critical) to streamline analysis.
- **Avoid overlogging:** Log only significant events according to

the domain and relevant data, avoiding unnecessary verbosity.

- **Data sensitivity:** Protect sensitive information by not including passwords or personal data in logs.

We can use Serilog to write information logs, which can be configured in **Program.cs** as follows:

```
1. var builder = WebApplication.CreateBuilder(args);
2. // Serilog configuration
3. Log.Logger = new LoggerConfiguration()
4.     .MinimumLevel.Debug() // Set minimum log level
5.     .WriteTo.Console() // Log to console
6.     .WriteTo.File("logs/log.txt"
7.         , rollingInterval: RollingInterval.Day)
8.     .CreateLogger();
9.
10. // Use Serilog as the logging provider
11. builder.Host.UseSerilog();
12.
13. var app = builder.Build();
14. app.UseRouting();
15. app.UseEndpoints(endpoints =>
16.     app.MapControllers();
17.
18. app.Run();
```

This basic setup can be extended to include additional sinks (such as databases, centralized logging systems, etc.) and more advanced configurations (such as filtering logs by source or level).

Error Notification System

In any software system, not all errors can be logged and calmly analyzed later. While logs provide valuable information to diagnose problems after they occur, some critical errors require immediate attention and may not wait for analysis after the accident. In these cases, it is essential to have a critical error reporting system that can alert a team to deal with and solve such problems. This proactive approach is crucial in systems where running time and reliability are essential, ensuring that critical problems are quickly identified

and mitigated to keep the system in perfect working condition.

An *Error Notification System* is a mechanism used to automatically notify developers or system administrators when critical errors or exceptions occur within the application. This system is crucial for proactive monitoring, allowing for quick detection and response to issues.

Serilog with notification sinks

We have seen Serilog as logging library. Serilog can be extended with various sinks that send notifications. For example, **Serilog.Sinks.Email** library can be used to send email notifications when errors occur. The configuration is quite easy, as follows:

```
1. await using var log = new LoggerConfiguration()
2.   .MinimumLevel.Error() // Log only errors
3.   .WriteTo.Email(
4.     {
5.       from = "sender@example.com",
6.       to = "receiver@example.com",
7.       host = "smtp.example.com"
8.     })
9.   .CreateLogger();
```

In this case, an email with the critical error will be send to team that can immediately fix the problem.

Best practices

Following are three best practices to implement an Error Notification System in an efficient way:

- **Notification filtering:** Configure the system to notify only for critical errors or relevant bugs to avoid excessive notifications that could lead to **noise**.
- **Integration with DevOps:** Link the Error Notification System with DevOps practices, such as continuous monitoring and incident management.
- **Sensitive information protection:** Ensure that notifications do not include sensitive information that could be exposed insecurely.

In summary, an Error Notification System in ASP.NET Core is a vital component for maintaining application stability and quality. By using appropriate tools and libraries, we can set up a robust system that helps in promptly addressing and resolving issues.

Post-mortem analysis

The **post-mortem analysis** is a crucial process undertaken after a significant incident or failure within an application. This analysis aims to thoroughly investigate the causes of the incident, document its impact, and outline steps for corrective actions and future prevention. In the context of ASP.NET Core applications, this involves utilizing tools like Serilog to gather comprehensive log data, as seen in previous paragraph, which provides detailed insights into what went wrong. The goal of a post-mortem analysis is not only to resolve the immediate issues but also to learn from them, improving the resilience and reliability of the application. By systematically examining the incident, development teams can identify weaknesses in the system, implement improvements, and ultimately deliver a more robust product.

Preparation and data collection with Serilog

This is the first phase of the process that begins by gathering all relevant data from the logs generated by Serilog. Ensure that the logs are comprehensive, including detailed information such as error messages and exceptions, the context of the exceptions including stack traces, request details like URL, parameters, headers, the involved user, and event timestamps. Utilize properties like **CorrelationId** to track the sequence of related events, allowing to reconstruct the application's flow up to the point of the error.

Incident description

Using the collected data, clearly describe the incident, specifying when it started and ended based on the log timestamps. Document the symptoms observed, such as errors displayed to users or functionality malfunctions, and assess the incident's impact in terms of the number of affected users, unavailable services, or other significant consequences.

Root cause analysis

Thoroughly analyze the logs to identify the root causes of the incident. Look for errors in the code by examining specific exceptions and their related stack traces, which can reveal bugs or logical errors. Review the logs to identify any configuration issues that may have contributed to the incident or to check for recent changes in settings. Additionally, investigate any reports in the logs about problems with external resources, such as databases or third-party services, that may have affected the incident.

Immediate corrective actions

Document the immediate corrective actions taken following the incident. This may include code fixes and patches to address identified bugs, adjustments to configurations to correct any errors, and specific actions to restore full application functionality. Be sure to record all changes made and the outcomes achieved.

Future improvements

Use the insights gained from the analysis to propose future improvements. This might involve implementing more robust exception handling to prevent unhandled exceptions in the future, as well as optimizing configurations to avoid similar issues. Also, consider integrating advanced monitoring and alerting tools by configuring additional Serilog sinks, such as *Application Insights* or other centralized monitoring systems, to enhance continuous monitoring of the application.

Documentation and sharing

Compile a detailed post-mortem report, including a comprehensive description of the incident, a root cause analysis with specific references to the Serilog logs, and a list of corrective actions and proposed improvements. Share this report with the development team and other stakeholders, ensuring it is stored in an accessible location for future reference. Normally a post-mortem report is written as follows:

Post-mortem report

Incident summary

Date and time: July 28, 2024, 14:30 - 15:15 UTC

Duration: 45 minutes

Affected systems: User authentication service

Impact: Approximately 5,000 users unable to log in or access their accounts.

Incident description

On July 28, 2024, the user authentication service experienced a significant outage. Users attempting to log in were met with a 500 Internal Server Error. The issue was first reported at 14:30 UTC and was resolved by 15:15 UTC. The outage resulted in approximately 5,000 users being unable to access their accounts during this period.

Root cause analysis

Upon investigation, it was determined that the root cause was a misconfiguration in the database connection settings. A recent deployment included an incorrect connection string pointing to a non-existent database instance. This misconfiguration led to a failure in the authentication process, as the service could not access the required user data.

Immediate corrective actions

- **Configuration rollback:** The incorrect connection string was quickly identified and rolled back to the correct configuration.
- **Service restart:** The authentication service was restarted to ensure that the correct settings were applied.
- **User communication:** Affected users were notified of the issue and informed of the resolution through email and a notification on the status page.

Recommendations and future preventive measures

- **Configuration management:** Implement stricter controls and validation for configuration changes, including automated tests to verify connection strings before deployment.
- **Monitoring enhancements:** Enhance monitoring and alerting for configuration-related issues, ensuring that incorrect settings are detected promptly.
- **Incident response training:** Conduct training sessions for the operations team to improve response times and incident management protocols.

Lessons learnt

This incident highlighted the importance of thorough testing and validation of all configuration changes. The team also recognized the need for better communication protocols to keep users informed during outages. By implementing the recommended actions, we aim to prevent similar incidents in the future and improve our overall service reliability.

Documentation and references

- **Change log:** Link to the deployment change log.

- **Incident timeline:** Detailed timeline of events and actions taken during the incident.
- **Logs and metrics:** Access to logs and metrics from Serilog and monitoring tools for further analysis.

Conclusion

The incident was resolved successfully within 45 minutes, with a clear understanding of the root cause and the steps needed to prevent future occurrences. The team will focus on implementing the recommended measures to enhance system stability and reliability.

Table 15.1: Post-mortem report

Review and feedback

Organize regular review meetings with the team to discuss the findings from the post-mortem and to review future actions. During this discussion, use the data from reports to provide concrete examples and specific details. This phase is crucial to ensure that all team members are aligned on the lessons learned and the necessary adjustments to improve future processes.

Exception handling

In any application, robust error and exception handling is crucial to ensure a smooth user experience and provide meaningful feedback to the developers. ASP.NET Core provides a variety of mechanisms for handling errors, ranging from simple try-catch blocks to more sophisticated middleware-based approaches.

Error versus exceptions

An **error** refers to a significant issue that occurs within a system, often leading a critical failure or problem that the application cannot easily recover from. Errors typically suggest conditions such as system malfunctions or server configuration problems, which may require intervention at a system level or specific configurations to resolve. Generally, errors are situations that cannot be managed or anticipated by the regular flow of the program's execution and may lead to malfunction or interruption of service.

An **exception** is an event that occurs during the execution of a program, disrupting the normal flow of operations. Unlike errors, exceptions are conditions that can be anticipated and handled by

the application. When an exception occurs, the application can catch it and handle it appropriately to prevent the application from crashing or behaving unexpectedly. Exceptions may arise from issues such as invalid input data, attempts to access unavailable resources, or logical errors within the application. In C#, exceptions can be managed using mechanisms like exception-handling middleware or **try-catch-finally** blocks.

While **errors** represent more severe and often unrecoverable issues that the application cannot handle, **exceptions** are manageable error conditions that can be addressed to maintain the application's stability and functionality.

Exceptions

In .NET, exceptions are represented by the **Exception** class and its derived classes. Common exceptions include **NullReferenceException**, **ArgumentException**, and **InvalidOperationException**.

When an exception occurs, the runtime searches for the nearest catch block that can handle the exception. If no catch block is found, the application terminates, potentially leading to a poor user experience.

This method involves wrapping potentially error-prone code within a try block and catching specific exceptions or general exceptions in one or more catch blocks, as follows:

```
1. try
2. {
3.     int result = 10 / int.Parse("0"); // Throw an exception
4. }
5. catch (DivideByZeroException ex)
6. {
7.     Console.WriteLine("Cannot divide by zero.");
8. }
9. catch (FormatException ex)
10. {
11.     Console.WriteLine("Invalid input format.");
12. }
13. catch (Exception ex)
14. {
```

```
15. Console.WriteLine("An error occurred: " + ex.Message);
16. }
```

While try-catch blocks are effective for handling known exceptions, they can become cumbersome when used extensively. For larger applications, a more centralized error handling mechanism is preferable.

Centralized error handling with middleware

ASP.NET Core's middleware pipeline allows create centralized error handling logic that can be applied globally to an application. This is typically done using the **UseExceptionHandler** middleware, which catches unhandled exceptions and routes them to a custom error-handling endpoint. In .NET 9, configuring the middleware is straightforward.

Following is how we can easily set up a global exception handler:

```
1. var builder = WebApplication.CreateBuilder(args);
2. var app = builder.Build();
3. app.UseExceptionHandler("/Error");
4. app.Map("/Error", async context =>
5. {
6.     context.Response.ContentType = "text/html";
7.     await context.Response
8.         .WriteAsync("<h1>An unexpected error occurred</h1>");
9. });
10. app.Run(async (context) =>
11. {
12.     // Simulating an error
13.     throw new Exception("Test exception");
14. });
15. app.Run();
```

In this example, any unhandled exceptions will be caught by the **UseExceptionHandler** middleware and routed to the `/Error` endpoint (line 3), which provides a simple HTML response. This approach ensures that all exceptions are handled gracefully without exposing sensitive details to the user.

Custom error pages

For a more user-friendly experience, especially in production environments, it is recommended to provide custom error pages. Custom error pages can be configured in the **UseExceptionHandler** middleware by specifying a path to a controller action or a static file that returns the error page. As we have seen in the previous section, we configured the middleware to use a properly endpoint

app.UseExceptionHandler("/Home/Error");

It will correspond to a controller as follows:

```
1. public class HomeController : Controller
2. {
3.     public IActionResult Error()
4.     {
5.         var exceptionDetails = HttpContext.Features.Get
6.             <IExceptionHandlerPathFeature>();
7.         if (exceptionDetails != null)
8.         {
9.             // Log details of error
10.            Log.Error(exceptionDetails.Error,
11.                "An error occurred at path {Path}"
12.                , exceptionDetails.Path);
13.        }
14.        return View();
15.    }
16. }
```

The correspondent view will be easily as follows:

```
1. @{
2.     ViewBag.Title = "Error";
3. }
4. <h1>Something went wrong</h1>
5. <p>Please try again later.</p>
```

In this setup, when an error occurs, the user is redirected to the **/Home/Error** endpoint, which could serve a fully customized error page.

Exception filters

In addition to middleware, ASP.NET Core provides exception filters, which can be used to handle exceptions thrown by controller actions. Exception filters can be globally configured or applied to specific controllers or actions.

To use an exception filter, we have to create a custom class that implements the **IExceptionHandler** interface. An example should be as follows:

```
1. public class CustomExceptionHandler : IExceptionHandler
2. {
3.     private readonly ILogger logger;
4.     public CustomExceptionHandler
5.         (ILogger<CustomExceptionHandler> logger)
6.     {
7.         this.logger = logger;
8.     }
9.     public void OnException(ExceptionContext context)
10.    {
11.        logger.LogError(context.Exception
12.            , "An unhandled exception occurred.");
13.        context.Result = new ContentResult
14.        {
15.            Content
16.                = "An error occurred while processing your
request.",
17.            StatusCode = 500
18.        };
19.        context.ExceptionHandled = true;
20.    }
21. }
```

This filter logs the exception and returns a simple error message. It can be registered globally or per-controller.

Following code applied in **Program.cs** applies the filter globally:

```
1. builder.Services.AddControllersWithViews(options =>
2. {
3.     options.Filters.Add<CustomExceptionHandler>();
4. });
```

Or else, apply it as an attribute to a specific controller as follows:

```
1. [ServiceFilter(typeof(CustomExceptionHandler))]
```

```
2. public class HomeController : Controller
3. {
4.     // Controller actions
5. }
```

Error and exception handling are fundamental aspects of building robust and reliable ASP.NET Core applications. By using a combination of try-catch-finally blocks, middleware, logging, and exception filters, we can effectively manage and respond to errors in our applications. .NET 9 simplifies the configuration process, allowing for a more streamlined and maintainable setup.

As we continue to build and deploy applications, we should always prioritize proper error handling to ensure a seamless experience for our users and valuable insights for our development team.

Monitoring and diagnostic tools

Monitoring and diagnostic tools are essential components in the lifecycle of an ASP.NET Core application, providing visibility into the application's performance, health, and potential issues. These tools enable developers and system administrators to track key metrics, such as response times, error rates, and resource usage, offering insights that can help in identifying performance bottlenecks or underlying problems. By capturing detailed logs and metrics, monitoring tools allow teams to understand how their application behaves under different conditions and loads. Diagnostic tools, on the other hand, delve deeper into the specifics of the application's operation, offering capabilities such as tracing, profiling, and debugging. They help in pinpointing the root causes of issues, whether they stem from code inefficiencies, memory leaks, or unexpected exceptions. Together, these tools play a critical role in maintaining the application's stability and reliability. They not only assist in resolving issues more swiftly but also in proactively identifying potential problems before they escalate into critical failures. The integration of comprehensive monitoring and diagnostics into the development and operational workflow allows for continuous improvement of the application, ensuring that it meets user expectations for performance and reliability.

In ASP.NET Core, several popular tools are commonly used for

monitoring and diagnostics, each offering distinct features to help developers manage application performance and troubleshoot issues.

Following are some key tools widely used in the ASP.NET Core ecosystem:

- **Application Insights** is a comprehensive service provided by Microsoft, offering real-time telemetry, analytics, and alerting capabilities. It integrates seamlessly with ASP.NET Core applications, providing insights into performance metrics, request tracking, error logging, dependency monitoring, and user behavior analysis.
- **Serilog** we know this tool as a structured logging library and is known for its flexibility. It supports various output sinks, allowing logs to be sent to different destinations like consoles, files, databases, or external services. This structured approach to logging simplifies the process of log analysis and querying.
- **Elastic Stack (ELK)** consisting of Elasticsearch, Logstash, and Kibana, is widely used for log aggregation and analysis. Elasticsearch provides powerful search and analytics capabilities, Logstash handles the ingestion and transformation of log data, and Kibana offers visualization tools to create dashboards and reports from log data.
- **Prometheus and Grafana** are often used together for metrics monitoring and visualization. Prometheus is an open-source system that collects metrics from configured endpoints, while Grafana provides a rich interface for visualizing these metrics through customizable dashboards. Together, they help in monitoring the health and performance of applications and infrastructure.
- **OpenTelemetry** is a newer standard that unifies the collection of telemetry data, including traces, metrics, and logs. It offers a set of APIs and libraries for capturing this data from applications and infrastructure, providing a consistent and open-source approach to observability.

These tools and frameworks provide essential capabilities for monitoring, diagnostics, and maintaining the overall health of

ASP.NET Core applications. We use these tools to gain insights into application behavior, detect issues early, and ensure high availability and performance. Moreover, they provide valuable data that informs our decisions about scaling, optimizing resource utilization, and enhancing the overall user experience. In essence, monitoring and diagnostic tools are indispensable for us in sustaining the operational excellence of modern web applications.

Conclusion

The chapter provided a comprehensive overview of essential practices for maintaining and enhancing the stability and reliability of ASP.NET Core applications. We explored logging and error tracking, emphasizing the importance of capturing detailed logs to understand and troubleshoot issues effectively. A crucial aspect highlighted was the post-mortem analysis, which plays a vital role in investigating incidents, understanding root causes, and preventing future occurrences. This analysis not only aids in resolving the immediate issues but also provides invaluable insights for continuous improvement. Through exception handling, we highlighted strategies to manage unexpected behaviors gracefully, ensuring that the application remains robust and user-friendly. Additionally, we delved into monitoring and diagnostic tools, showcasing their crucial role in proactively identifying and addressing performance issues and other potential problems. Together, these topics underscore the importance of a holistic approach to application maintenance, where consistent monitoring, thorough logging, strategic error handling, and insightful post-mortem analyses converge to create a resilient and reliable system. By implementing these best practices, developers can quickly resolve issues and enhance user experiences through a deeper understanding of their applications.

Points to remember

- **Centralized logging:** Ensures consistent error tracking across the application.
- **Exception filters:** Customize error responses and streamline exception handling.

- **Global exception handling:** Use middleware for unified error management.
- **Detailed logging:** Includes user actions for comprehensive issue tracing.

Multiple choice questions

1. **What is the primary purpose of logging in an ASP.NET Core application?**
 - a. To increase application performance
 - b. To monitor and track application behavior
 - c. To reduce memory consumption
 - d. To automatically fix bugs
2. **What is meant by exception handling?**
 - a. Automatically fixing errors
 - b. Writing custom messages in logs
 - c. Capturing and managing errors to prevent crashes
 - d. Improving application performance
3. **Which method in ASP.NET Core can be used for global exception handling?**
 - a. UseExceptionHandler
 - b. UseLogging
 - c. UseStaticFiles
 - d. UseAuthorization
4. **What is a common practice when handling exceptions in an ASP.NET Core application?**
 - a. Ignoring them
 - b. Logging them for analysis
 - c. Only handling them in the user interface
 - d. Disabling exception handling for performance reasons
5. **What is the primary purpose of a post-mortem analysis in**

software development?

- a. To celebrate the completion of a project
- b. To document and understand the causes of an incident
- c. To plan for the next project
- d. To evaluate the financial success of the project

6. During a post-mortem analysis, what is the significance of identifying a root cause?

- a. To blame the team members responsible
- b. To find the initial issue that led to the incident
- c. To decide who will lead the next project
- d. To determine the project's financial outcome

Answer key

Key terms

- **Centralized logging:** Standardizes error tracking across the entire application.
- **Exception filters:** Customize error responses and manage exceptions efficiently.
- **Global exception handling:** Middleware to manage errors across all requests.
- **Custom error pages:** Enhance user experience by providing meaningful error information.
- **Error Notification Systems:** Alerts teams to issues for quicker response and resolution.

CHAPTER 16

Containerization for Seamless Deployment

Introduction

Containerization has revolutionized the way applications are developed and deployed, enabling the creation of isolated execution environments that work consistently across various platforms. In this chapter, we will explore the fundamental concepts of containerization and its impact on modern software development. We will start with an introduction to containerization, highlighting its benefits over traditional virtual machines. Then, we will delve into Docker, the most popular containerization platform, and how it integrates with ASP.NET Core for efficient web application development. Next, we will explore container orchestration using Kubernetes, a powerful system for managing containers at scale, ensuring high availability and automated management. We will conclude by discussing the handling of configuration and dependencies, offering strategies to address these challenges securely and effectively.

This chapter provides a comprehensive overview of how containerization can streamline the deployment process, making it

more seamless and reliable.

Structure

In this chapter, we will cover the following topics:

- Introduction to containerization
- Docker and ASP.NET Core integration
- Orchestrating containers with Kubernetes
- Handling configuration and dependencies

Objectives

By the end of this chapter, readers will have a comprehensive understanding of containerization and its practical application. The chapter aims to explain the basic principles of containerization and highlight its benefits compared to traditional virtualization methods. It also focuses on the integration of Docker with ASP.NET Core, demonstrating how to efficiently develop and deploy web applications using containers. Additionally, we will understand what Kubernetes is and how to use it for orchestrating containers, emphasizing its features for scaling, self-healing, and service discovery.

Introduction to containerization

Containerization is a lightweight form of virtualization that involves encapsulating an application and its dependencies into a container. This container is an isolated, portable environment that ensures the application runs consistently across different computing environments, such as development, testing, and production. Unlike traditional virtual machines, containers share the host system's kernel but maintain isolation at the process level, which makes them more efficient in terms of resource utilization.

Microservices architecture

We have already encountered this kind of architecture in [Chapter 3, Architecture and Core Components](#). We defined it as follows:

Microservices architecture decomposes an application into a set of

small, independent services, each responsible for a specific function. This promotes scalability and flexibility but requires careful management of inter-service communication.

The key principles of this architecture include service independence, where each service is autonomous and communicates with others through well-defined APIs. Resilience is also crucial, with each service designed to be fault-tolerant, ensuring that an issue in one service does not affect the entire system. Additionally, microservices architecture emphasizes scalability, allowing critical services to be horizontally scaled based on demand.

This approach promotes a faster development cycle, the adoption of diverse technologies for different services, and more efficient resource management.

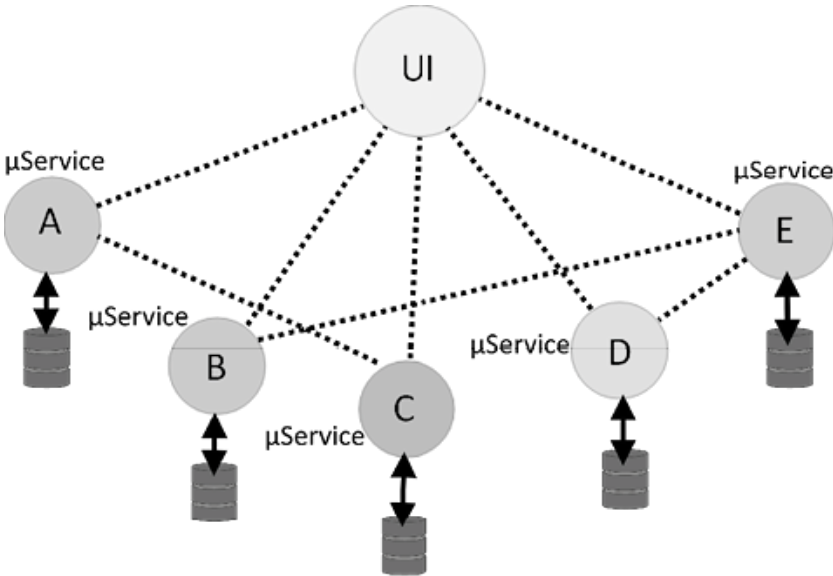


Figure 16.1: Microservices architecture

Docker is one of the possible players in implementing microservices architecture (perhaps the most widely used), as it allows for the containerization and isolation of individual application components. With Docker, each microservice can be developed, tested, and deployed independently, ensuring that dependencies and configurations are managed autonomously without interfering with other services. This technology facilitates scalability, allowing for

multiple instances of a service to run on different machines or clouds. Additionally, Docker containers make it easy to replicate production environments, enhancing efficiency and predictability in the development and deployment process.

Key benefits of containerization

Containerization provides various benefits that can significantly enhance the development and deployment of applications.

Following are some of the key advantages that can be achieved with this technology:

- **Consistency across environments:** Containers ensure that the application behaves the same way, regardless of where it is deployed, reducing the “**It works on my machine**” problem.
- **Scalability and efficiency:** Containers can be easily scaled up or down, and they consume fewer resources compared to virtual machines because they share the host’s operating system kernel.
- **Isolation:** Containers isolate applications from one another and from the host system, enhancing security and stability.

Due to the above advantages, we can affirm that containerization is a great tool for modern software development, offering efficiency and reliability.

Steps to achieve containerization

Containerizing an ASP.NET Core application involves several essential steps to ensure a consistent and efficient deployment process.

Following is a breakdown of the key steps:

1. **Identifying application and dependencies:** including necessary libraries, runtime, and configuration files specific to ASP.NET Core.
2. **Dockerfile:** Once these elements are outlined, we can create a Dockerfile. This file contains instructions for building a Docker image, specifying a base image compatible with ASP.NET Core, such as an official ASP.NET Core runtime

image. The Dockerfile also copies the application files, restores dependencies using the .NET CLI, and sets environment variables essential for the application's operation.

3. **Build the Docker image:** Use the **command line interface (CLI)** to build the Docker image from the Dockerfile. This image encapsulates the ASP.NET Core application along with its runtime environment, ensuring consistency across different deployment stages.
4. **Run the container:** After building the image, it can be run as a container, providing an isolated environment where the ASP.NET Core application operates independently.
5. **Kubernetes:** For more complex deployments involving multiple instances or additional services, Kubernetes can be utilized to manage and orchestrate the deployment, scaling, and monitoring of ASP.NET Core containers. This orchestration facilitates smooth and automated management, enhancing the scalability and reliability of the application in a production environment.

Install Docker Desktop

If we want to containerize an ASP.NET Core application, we have to install Docker Desktop. Docker is licensed under a mix of open-source and commercial agreements, such as the Apache 2.0 License for Core Components and the Docker Subscription Service Agreement for commercial use. Careful attention is required to ensure compliance and avoid licensing violations, especially in business environments. To install Docker Desktop, first visit the Docker website and download the appropriate version for the operating system we want to install on, either Windows or macOS. For Windows users, double-click the downloaded installer file (.exe) and follow the on-screen instructions, granting administrator privileges when prompted. Once the installation is complete, Docker Desktop will automatically launch.

For macOS users, download the Docker Desktop disk image (.dmg) file, open it, and drag the Docker icon to the Applications folder. After that, open Docker from the applications folder, and we may

need to provide the password to authorize the installation of a helper tool.

Once Docker Desktop is installed, we can launch it, and the Docker whale icon will appear in the taskbar (Windows) or menu bar (macOS). Docker Desktop will take a few moments to start, and once running, it will be ready for use, allowing us to manage Docker containers and images on our local machine.

Docker and ASP.NET Core integration

Docker allows developers to package applications with all necessary dependencies into a container, ensuring consistent behavior across different environments. This integration simplifies deployment and scaling, enabling ASP.NET Core applications to run efficiently on various platforms, from local development machines to cloud infrastructure.

The Dockerfiles

A Dockerfile is a script composed of a series of instructions that describe how to build a Docker image. It is essentially a blueprint for creating a container, detailing everything from the base image to be used, to the installation of necessary packages, and the configuration of the application environment.

A typical Dockerfile begins with a **FROM** instruction specifying the base image, such as **mcr.microsoft.com/dotnet/aspnet:8.0** for an ASP.NET Core application. This is followed by **COPY** commands to transfer application files into the container, and **RUN** commands to execute commands like installing dependencies. To use a Dockerfile, it is placed in the root directory of the application, and the **docker build** command is executed to create an image. This image can then be run as a container, ensuring that the application runs consistently across different environments.

Create a containerized website

To create a containerized website allows for the isolation of the execution environment, ensuring consistency across development, testing, and production. This approach offers the flexibility to easily manage dependencies and configurations, making application

deployment and scaling more straightforward. However, the complexity in setting up and managing containers can present challenges, requiring specific skills. Additionally, resource overhead may increase compared to running directly on the operating system. Despite these drawbacks, the benefits of portability, scalability, and isolation make containers a popular choice for modern web development.

To containerize an ASP.NET Core application, we have to write the Dockerfile and place it in the root directory of our project. The Dockerfile will be similar to the following:

```
1. # Use the official ASP.NET Core runtime as a parent image
2. FROM mcr.microsoft.com/dotnet/aspnet:9.0 AS base
3. WORKDIR /app
4. EXPOSE 8080
5. FROM mcr.microsoft.com/dotnet/sdk:9.0 AS build
6. ARG BUILD_CONFIGURATION = Release
7. WORKDIR /src
8. COPY Chapter16Docker.csproj ./
9. RUN dotnet restore "Chapter16Docker.csproj"
10. COPY . .
11. WORKDIR "/src"
12. RUN dotnet build "Chapter16Docker.csproj" -c
    $BUILD_CONFIGURATION -o /app/build
13. FROM build AS publish
14. ARG BUILD_CONFIGURATION = Release
15. RUN dotnet publish "Chapter16Docker.csproj" -c
    $BUILD_CONFIGURATION -o /app/publish /
    p:UseAppHost=false
16. FROM base AS final
17. WORKDIR /app
18. COPY --from=publish /app/publish .
19. ENTRYPOINT ["dotnet", "Chapter16Docker.dll"]
```

When the Dockerfile is ready, it will be passed to the Docker engine to pack it up in a proper image and it will be ready to be used as a container or distributed. From a command prompt, we can run the command **docker build -t my-aspnetcore-app** (do not forget the

point at the end of the command) to build the container image of our application. The output will be as follows:

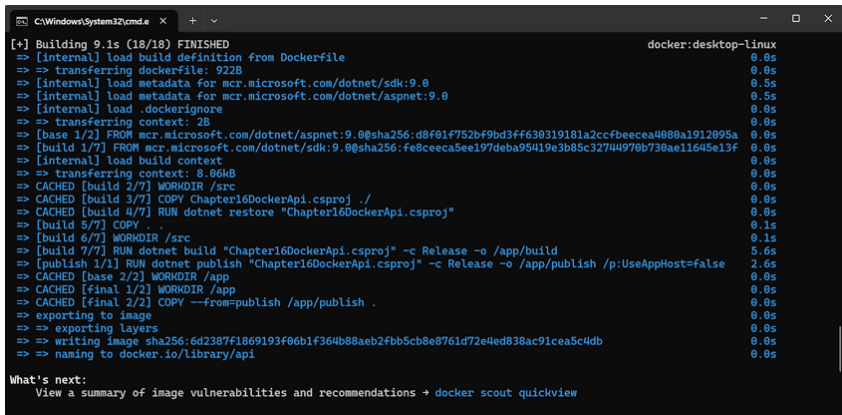


Figure 16.2: Docker command output

Then, we can find our image in the Docker Desktop environment as follows:

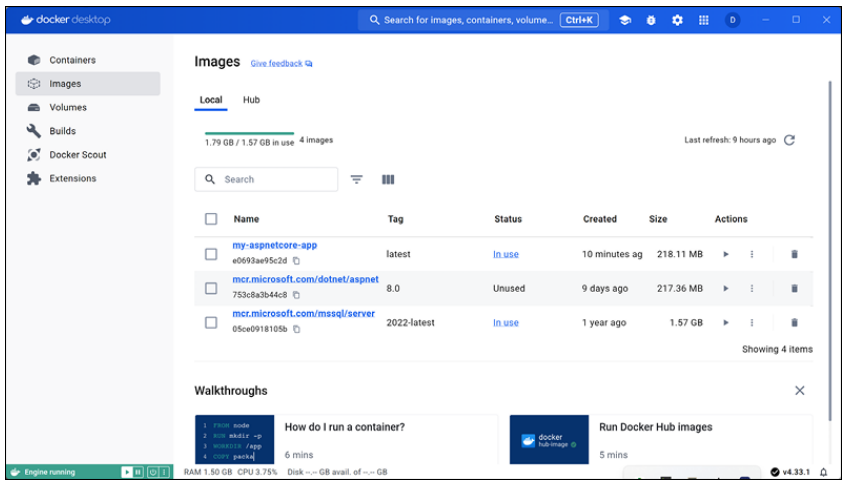


Figure 16.3: Docker image

The last step is to start a container based on our Docker image on a specific port and with a specific name. The command will be as following:

docker run -d -p 8080:8080 --name app-core my-aspnetcore-app

This command line is used to start a Docker container with specific

options.

Following is a breakdown of what each part of the command does:

- **docker run**: This is the base command used to create and start a new container from a specified image.
- **-d**: This flag runs the container in detached mode, meaning the container runs in the background and does not block the terminal.
- **-p 8080:8080**: This option maps port 8080 on the host (the machine where the Docker daemon is running) to port 8080 on the container. This allows external access to the container's application through port 8080 on the host.
- **--name app-core**: This option assigns a name to the container, in this case, **app-core**. This makes it easier to reference the container in future Docker commands (e.g., **docker stop app-core**, **docker logs app-core**, etc.).
- **my-aspnetcore-app**: This is the name of the Docker image from which the container is created. In this case, it is assumed to be an ASP.NET Core application. The image must be available locally or in a Docker registry accessible to our environment.

The final output of this command is the container on our Docker Desktop and our application will be available at the address <http://localhost:8080> as follows:

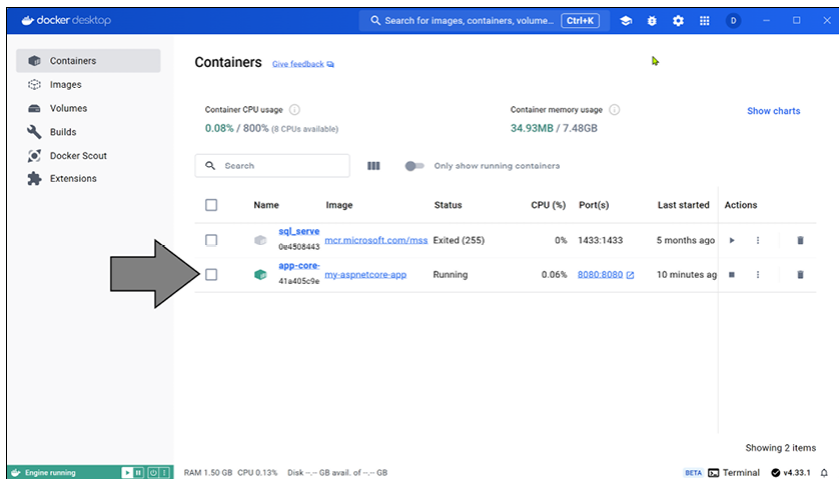


Figure 16.4: Docker container

The arrow shows our container up and running. Now we can open a browser and navigate to the localhost address to the specific port declared (in this case 8080). We will see our website as usual but it will be containerized.

Containerizing Web APIs

After containerizing our website, we can do the same with a Web API project, allowing us to have a fully containerized resource. This process involves packaging the backend services into containers, ensuring they run consistently in different environments.

Similarly to our website seen previously, we have to craft properly a Dockerfile. An example should be as follows:

1. # Use the official ASP.NET Core runtime as a parent image
2. FROM mcr.microsoft.com/dotnet/aspnet:9.0 AS base
3. WORKDIR /app
4. EXPOSE 8080
- 5.
6. # Use the SDK image to build and publish the app
7. FROM mcr.microsoft.com/dotnet/sdk:9.0 AS build
8. ARG BUILD_CONFIGURATION = Release
9. WORKDIR /src
10. COPY Chapter16DockerApi.csproj ./
11. RUN dotnet restore "Chapter16DockerApi.csproj"

- 12.
13. *# Copy the remaining files and build the application*
14. COPY . .
15. WORKDIR `"/src"`
16. RUN `dotnet build "Chapter16DockerApi.csproj" -c $BUILD_CONFIGURATION -o /app/build`
- 17.
18. *# Publish the application*
19. FROM `build AS publish`
20. ARG BUILD_CONFIGURATION = `Release`
21. RUN `dotnet publish "Chapter16DockerApi.csproj" -c $BUILD_CONFIGURATION -o /app/publish /p:UseAppHost=false`
- 22.
23. *# Use the base image to run the app*
24. FROM `base AS final`
25. WORKDIR `/app`
26. COPY `--from=publish /app/publish .`
27. ENTRYPOINT `["dotnet", "Chapter16DockerApi.dll"]`

As usual we build the image with the Docker command **docker build -t api .** where **api** will be the name of our image. Now we have to build our container with the following command **docker run -p 8080:8080 -d --name app-core-api api** so that we can have a proper container able to expose the web API properly on 8080 ports. After creating the container, we will see it in the list of containers in Docker Docker as follows:

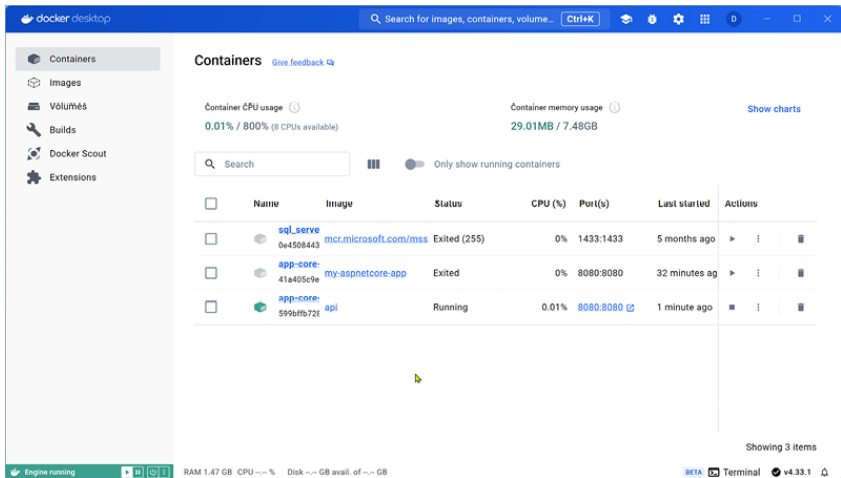


Figure 16.5: Web API container

To test our API, we can simply open a browser and type the address <http://localhost:8080/weatherforecast> then, we will see the following result:

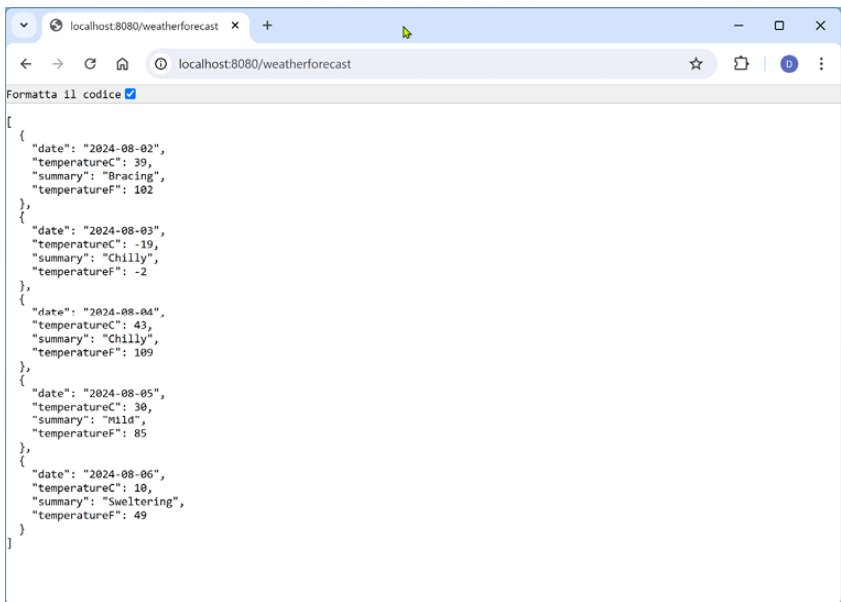


Figure 16.6: Web API result

After containerizing all our applications as website, Web API, database and so on, we can put all together and really build a real

application. Be sure that all the containers have a different port, instead an error will be thrown starting the container. Now, the problem should make them work properly and orchestrate all together.

Orchestrating containers with Kubernetes

Kubernetes (K8s) is a system for orchestrating Docker containers, allowing for efficient management and scaling of applications. A Docker container is like a package containing an application and all its dependencies. The steps to create a Kubernetes application are as follows:

- **Define a Pod for each Docker container:** A Pod is the basic unit in Kubernetes, capable of containing one or more containers. For each Docker image, we create a **YAML** configuration file to define the Pod, specifying which Docker image to use.
- **Create a deployment to manage the Pods:** A Deployment is a Kubernetes resource that ensures the desired number of Pod replicas are always running. It manages the lifecycle of the Pods. We use a **YAML** file to define the Deployment, associating Pods with their respective Docker images.
- **Services for exposure:** A service in Kubernetes exposes the Pods, making them accessible to each other and, optionally, to the outside world. We configure a service for each application to allow external access, specifying the ports and service type (such as load balancer for internet exposure).

For our examples, let us consider three generic Docker images: **app1**, **app2**, and **app3**. Each image will have an associated Deployment and Service.

Deployment files

A Deployment file in Kubernetes is a text file written in **YAML** language that defines the configuration of the Pods and their containers. It is used to state how applications should be deployed and managed, specifying the container image, the desired number of replicas, resource requirements, and upgrade policies. The Deployment file will be as follows:

1. `apiVersion: apps/v1`
2. `kind: Deployment`
3. `metadata:`
4. `name: app1-deployment`
5. `spec:`
6. `replicas: 1`
7. `selector:`
8. `matchLabels:`
9. `app: app1`
10. `template:`
11. `metadata:`
12. `labels:`
13. `app: app1`
14. `spec:`
15. `containers:`
16. `- name: app1-container`
17. `image: app1:latest`
18. `ports:`
19. `- containerPort: 80`

For the **app2** and **app3** it is the same **YAML** file but with different names.

Services files

A service file in Kubernetes is a structured **YAML** text file that defines how to expose an application running on a set of Pods. It specifies the service type (such as ClusterIP, NodePort, or LoadBalancer), the selector labels to match the Pods, and the ports for accessing the application. This configuration enables stable network endpoints and load balancing for the Pods.

Following is an example:

1. `apiVersion: v1`
2. `kind: Service`
3. `metadata:`
4. `name: app1-service`
5. `spec:`
6. `selector:`
7. `app: app1`

8. **ports:**
9. - **protocol:** TCP
10. **port:** 80
11. **targetPort:** 80
12. **type:** LoadBalancer

This **YAML** file sets up the exposure of the **app1** application on the internet. The configurations for **app2** and **app3** will follow a similar structure, with changes in the names and images.

How it works

The Deployment manages the lifecycle of the Pods, ensuring that a specified number of Pods are always running. If a Pod crashes, Kubernetes automatically restarts it. Pods contain Docker containers and are the execution units in Kubernetes. The Service acts as an entry point for the applications, enabling external access and communication between Pods.

How to use

To use this setup, ensure to have a working Kubernetes cluster. Once set up, we can apply the **YAML** files using the command **kubectl apply -f <filename>.yaml**, which will create and manage the resources in the Kubernetes cluster. This process allows to deploy and manage Docker containers efficiently with Kubernetes.

This orchestration allows for high availability, scalability, and efficient management of the different parts of the application, providing a robust infrastructure for running complex web applications.

Handling configuration and dependencies

Handling configuration and dependencies in a containerized environment ensures our applications run smoothly and securely. In a traditional deployment, configuration files and dependency management might be tightly coupled with the application code and the host environment. However, in containerization, these elements need to be abstracted and managed separately to maintain the container's portability and consistency across different environments.

One common approach for managing configuration is through the use of environment variables, which can be set at runtime and provide a flexible way to adjust settings without modifying the container image. This method is particularly useful for sensitive information such as database connection strings or API keys, as it keeps these details out of the application code and Dockerfiles.

Dependencies are typically handled by including them within the Docker image itself. This is achieved by specifying all necessary dependencies in the Dockerfile, ensuring they are installed during the build process. For example, in an ASP.NET Core application, the necessary .NET runtime and libraries would be restored and included as part of the image build.

For more complex configurations and sensitive data, Kubernetes ConfigMaps and Secrets can be utilized. ConfigMaps allow for the storage of non-sensitive configuration data, while Secrets are used for managing sensitive information. Both can be injected into containers at runtime, providing a secure and manageable way to handle configuration and dependencies. This separation of configuration and code promotes best practices in security and operational management, ensuring that applications can be easily updated and maintained without risking exposure of sensitive data.

Conclusion

The chapter provided a comprehensive overview of containerization and its pivotal role in modern software development. It covered the fundamental concepts and benefits of containerization, explored the integration of ASP.NET Core with Docker, and detailed the orchestration capabilities offered by Kubernetes. Additionally, the chapter discussed best practices for managing configuration and dependencies within containers, emphasizing the importance of secure and efficient management. By leveraging these technologies, developers can achieve consistent and reliable deployments across various environments. The insights gained from this chapter equip organizations with the knowledge to streamline their development processes, enhance application scalability, and ensure robust performance in production settings. As containerization continues to be a cornerstone of DevOps practices, mastering these tools and

techniques is crucial for staying competitive and delivering high-quality software solutions.

In the next chapter, we will introduce Blazor, exploring what it is and how it works. We will uncover its capabilities for building interactive web applications using C#, bridging the gap between client-side and server-side development.

Points to remember

- **Docker simplifies container management:** Provides tools for creating, deploying, and running containers efficiently.
- **Kubernetes provides scalability and management:** Automates the deployment, scaling, and management of containerized applications, making it easier to handle large-scale systems.
- **Dockerfile defines container environment:** A crucial component for specifying the setup and dependencies of Docker containers.

Multiple choice questions

1. **What is one primary benefit of containerization over traditional virtual machines?**
 - a. More extensive hardware requirements
 - b. Better isolation of applications
 - c. Higher resource overhead
 - d. Slower deployment times
2. **How does Docker integrate with ASP.NET Core?**
 - a. By using Dockerfiles to define application environments
 - b. By requiring a separate server for ASP.NET Core
 - c. By running applications directly on the host OS
 - d. By using ASP.NET Core's built-in containerization features
3. **What is a primary feature of Kubernetes?**
 - a. Manual scaling of applications

- b. Orchestration of virtual machines
- c. Automatic service discovery and load balancing
- d. Exclusive use with Java applications

4. What is a Pod in Kubernetes?

- a. The smallest deployable unit that can contain one or more containers
- b. A virtual machine running in a Kubernetes cluster
- c. A configuration file used to describe the desired state of the cluster
- d. A network layer component that routes traffic between services

Answer key

Key terms

- **Containerization:** The process of packaging an application and its dependencies into a container to ensure consistent environments across different stages of development and deployment.
- **Docker:** A popular platform for developing, shipping, and running applications inside containers.
- **Kubernetes:** An open-source platform for automating the deployment, scaling, and management of containerized applications.
- **Orchestration:** The automated arrangement, coordination, and management of computer systems, middleware, and services.
- **Dockerfile:** A text file that contains instructions to build a Docker image, specifying the environment and dependencies for an application.
- **Microservices architecture:** An architectural style that structures an application as a collection of small, autonomous services modeled around a business domain.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 17

Building Responsive User Interfaces with Blazor

Introduction

In this chapter, we will learn the fundamentals of Blazor, an innovative framework for building interactive web applications with .NET. We will start by defining what Blazor is, highlighting its core features and benefits. We will then delve into the architecture for Blazor applications, providing insights into its components and design paradigms. We will differentiate between Blazor Server and Blazor WebAssembly, explaining the two hosting models and how they impact application performance and deployment. Lastly, we will introduce the concept of Blazor full stack web UI introduced in .NET 8. This section will explain how Blazor full stack web UI has improved the performances of Blazor applications.

Structure

In this chapter, we will cover the following topics:

- What is Blazor
- Blazor Server
- Blazor Webassembly
- Blazor Hybrid
- Blazor render modes

Objectives

By the end of this chapter, readers will be provided a thorough understanding of Blazor, exploring what it is and its distinctive features. We will delve into the architecture of Blazor applications, examining the various components and their roles within web development. Our aim is also to clarify the differences between Blazor Server and Blazor WebAssembly, discussing their respective advantages and limitations in terms of performance and deployment. Additionally, we will introduce the concept of **server-side rendering (SSR)**, **streaming rendering (SR)**, interactive server rendering and WebAssembly rendering. All the modalities concur to achieve the Blazor full stack web UI. By the end of this chapter, we aim to provide a comprehensive understanding of Blazor's capabilities and how to leverage them to build modern, efficient web applications.

What is Blazor

In recent years, the software development landscape has seen a significant shift from traditional desktop applications to web-based solutions. Initially driven by the need for cross-platform compatibility and simplified update management, this migration has revealed a host of advantages that make web applications increasingly attractive, even in traditionally standalone contexts applications.

Universal compatibility and accessibility

One of the main strengths of web applications is their universal compatibility. Unlike desktop applications, which often require specific hardware and software infrastructure, web applications can run on any device with a browser and an internet connection. This

eliminates the need to tailor the application for different operating systems or hardware configurations. However, for standalone applications that prefer not to use a traditional web browser, there are other options available. For instance, **Progressive Web Apps (PWA)** offer a native app-like experience with the ability to be installed directly on a device and work offline (when connectivity is not required). **Cross-platform native applications** developed using technologies like *.NET MAUI* enable the creation of native applications for multiple operating systems from a single codebase, while still leveraging the unique features of each platform. Another option is **WebView2**, a control for WPF and Windows Form applications provided by Microsoft that allows embedding web content within traditional or legacy desktop applications able to start the migration of new features to a web application in order to slowly migrate the entire application by modernizing it and rewriting the most critical features. WebView2 is based on Microsoft Edge (Chromium) and enables the integration of modern web technologies into traditional desktop applications. The advantages of WebView2 include access to the security and performance features of a modern browser, the ability to update web content without requiring a full application update, and seamless integration between web user interfaces and native platform functionalities. This is particularly useful for standalone software for kiosks, totems or machine tools, as it allows combining the interactivity and flexibility of web interfaces with the reliability and performance of desktop applications.

Simplified updates and maintenance

Desktop applications typically require manual updates, which can be complex and time-consuming, especially in industrial settings where downtime must be minimized. In contrast, web-based applications allow for centralized updates and new feature implementations. Users automatically access the latest version of the application without manual intervention, ensuring that all stations are consistently up to date. This is particularly advantageous for applications used in the context of interconnected computers, where operational efficiency and security are really important. For instance, on a machine tool or kiosk we could have

more than a cockpit (as independent PC) and applications able to work together. Use a traditional desktop-based application will be challenged because they will be synchronized and updated in the meantime with a big effort (we cannot update in a single executable but with multiple updates for each involved PC). With a web-based application, we have a single application connected on the same backend that will provide always updated pages. When an update of the application will be available, it will be necessary to update the server part of our web application and everything will be updated.

Remote access and collaboration

Another significant advantage of web applications is the ability to access data and controls remotely. This is particularly useful for machine tools, where operators and managers can monitor and manage operations from different locations. The ability to access applications via mobile devices allows engineers and technicians to respond quickly to issues, improving responsiveness and reducing downtime. Additionally, collaboration among teams in different geographical locations is simplified, thanks to shared access to the same resources and real-time data.

A web-based solution for .NET

In this context of increasing adoption of web applications, Blazor emerges as an ideal solution for .NET developers. Blazor enables the creation of **single page applications (SPAs)** using C# and the .NET framework, bypassing the need for complex transitions to JavaScript. With Blazor, developers can build robust, interactive web applications that can easily replace traditional desktop applications, even in complex sectors like machine tools.

Blazor is closely integrated with ASP.NET Core, providing a modern way to develop web applications using the .NET framework. Blazor uses the Razor component model, a technology familiar to ASP.NET Core developers, to build interactive and dynamic user interfaces. This integration allows for the sharing of business logic, data models, and services between traditional server-side ASP.NET Core applications and modern SPA.

Blazor supports two main hosting models: **Blazor Server**, which performs processing on the server and uses SignalR to communicate

updates to the client (we spoke about SignalR in [Chapter 11](#), *SignalR in Realtime Applications*), and **Blazor WebAssembly**, which enables running the application directly in the browser via WebAssembly. This flexibility allows developers to choose the approach that best suits the specific needs of their project, while leveraging the robust capabilities of ASP.NET Core for handling security, authentication, and data access.

The Razor syntax rules

The need for a language like Razor arose from the desire to simplify and enhance the web development process for .NET developers. Traditionally, creating dynamic web content required using separate languages for server-side business logic (such as C#) and client-side markup and presentation (like HTML and JavaScript). This separation often led to a fragmented codebase, making project management and maintenance more challenging.

Razor was designed to integrate C# and HTML into a single syntax, allowing developers to write server-side code and HTML markup in the same file. This approach not only simplifies the integration between the server and client sides but also reduces development complexity and time, thus improving productivity. It enables the creation of more responsive and dynamic web applications by allowing efficient updates to the user interface. Essentially, Razor addresses the need for a unified and intuitive framework for building modern, interactive web applications.

Razor syntax is the base of Blazor. Following is the basic of the Razor syntax inherited from Blazor:

- **Code blocks:** C# code within a Razor page is enclosed in `@{ ... }` blocks.

Following is an example:

```
1. @{
2.     var message = "Hello, world!";
3.     var currentTime = DateTime.Now;
4. }
5. <p>@message The current time is @currentTime.</p>
```

- **Inline expressions:** Inline C# expressions are prefixed with

@. Here follows a function in C# as an example:

```
1. public string GetGreeting() {  
2.     return "Welcome!";  
3. }
```

If we want to call the function from a Razor (Blazor) page, will be as follows:

```
1. <p>@GetGreeting()</p>
```

- **HTML content:** Razor pages can contain standard HTML markup.
- **Razor comments:** Razor comments are denoted with @* ... *@ and are not rendered in the output.

Following is an example:

```
1. @* This is a Razor comment *@
```

- **Control statements:** Control flow statements (if, for, foreach) are used as in C# but with @ to indicate Razor code.

Following is an example:

```
1. @if (true)  
2. {  
3.     <p>This is rendered if true.</p>  
4. }  
5. @foreach (var number in numbers)  
6. {  
7.     <p>Number: @number</p>  
8. }
```

- **One-way binding:** Variables and model properties are accessed using @. This process is also called **one-way binding**.

Following is an example:

```
1. <p>Name: @person.Name</p>  
2. <p>Surname: @person.Surname</p>  
3. <p>Age: @person.Age</p>
```

- **Two-way binding:** allows for both the UI and the data model

to be updated based on changes in each other. This is particularly useful for form inputs and other interactive elements where user input can modify the underlying data. When a user changes the value in a form field, the corresponding property in the model is updated, and any subsequent changes to the model will be reflected in the UI. This dual synchronization simplifies the handling of user inputs and data management in interactive applications.

Following is an example:

```
1. <input type="text" @bind="person.Name" />
```

This is the simplest way to bind a property to a field of text, but we can have more functionalities as **@bind-Value:get="Name"** or **@bind-Value:set="(value) => { Name = value; }"** that allows us to detach the get and set of the property. We can also use **@bind:after="() => { }"** or **@bind-Value:after="() => { }"** to perform Actions after the assign of the value.

- **Directives:** Razor uses directives like **@page**, **@using**, **@inherits**, etc., for various configurations.

Following is an example:

```
1. @page "/example"  
2. @using System.Collections.Generic  
3. @inherits LayoutComponentBase
```

- **Functions and methods:** Functions and methods can be defined using **@code** directive. This section will produce a partial class at compile time. We can also create by ourselves this partial class adding a file in the same directory of our razor component or page with the same name and with **.razor.cs** extension. In Visual Studio the file **.razor** and the **.razor.cs** with the same name files are visualized together.

The partial class will be as follows:

```
1. public partial class Weather  
2. {  
3.     public string Name { get; set; }  
4.     private WeatherForecast[]? forecasts;
```

5. ...

The Visual Studio solution explorer will show the files as follows:

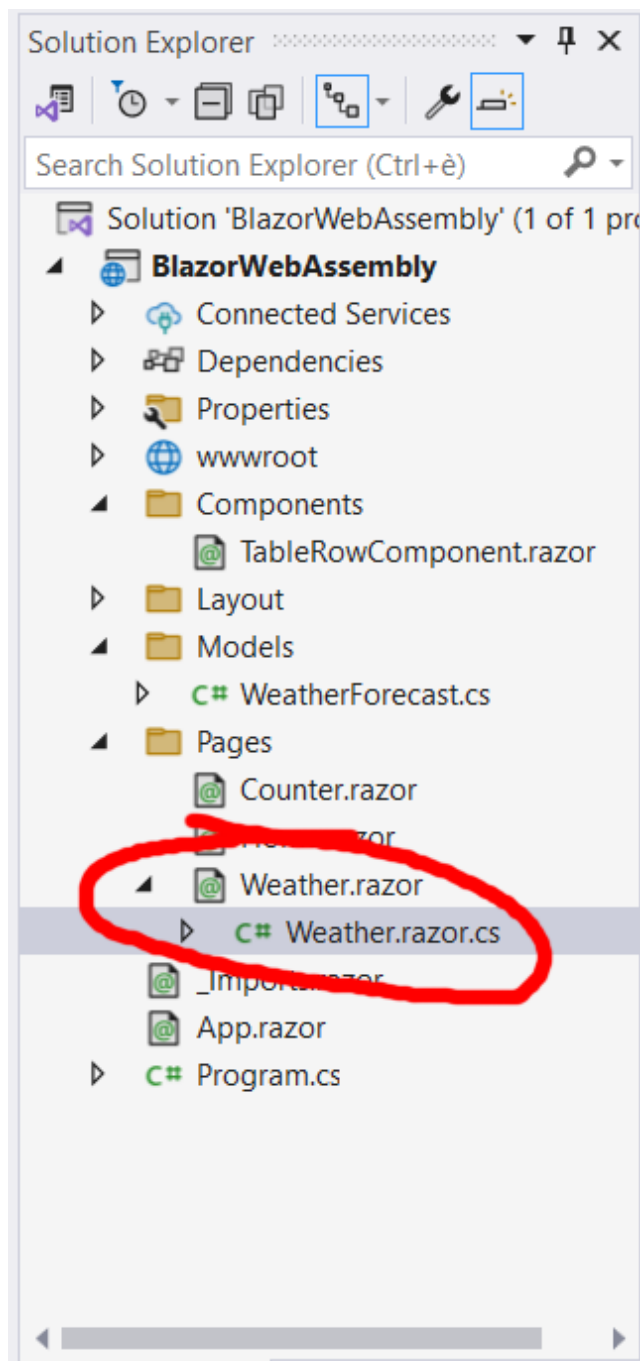


Figure 17.1: Visual Studio partial razor class

Blazor Server

Blazor Server, introduced with ASP.NET Core in 2018, enables the creation of interactive web applications using primarily C# instead of JavaScript (we must use JavaScript for all those critical features for browser that only JavaScript is authorized to use). In Blazor Server, the application logic runs on the server, and UI updates are sent to the client via real-time communication. Following is a simplified diagram of how Blazor Server works:

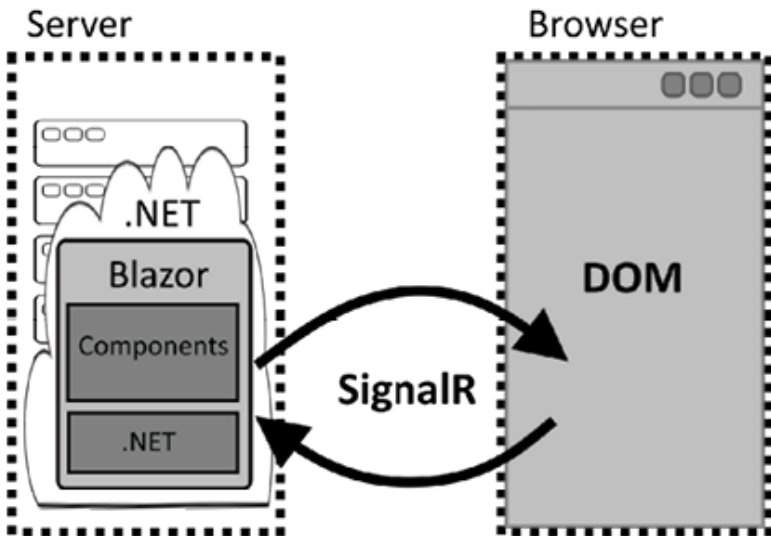


Figure 17.2: Blazor Server

This unique model has several key features and considerations, explained in detail in the next section.

Architecture and communication model

In Blazor Server, the application components, including their state and logic, are managed on the server. The client-side UI is rendered in the user's browser, while all interactions and events are processed server-side. Communication between the client and server is handled via SignalR, which maintains a persistent connection to send and receive updates.

For example, consider a simple counter page included in Blazor Server template. The state and increment logic reside on the server,

as we can see in the following code:

```
1. @page "/counter"
2. <h3>Counter</h3>
3. <p>Current count: @currentCount</p>
4. <button @onclick="IncrementCount">Increment</button>
5. @code
6. {
7.     private int currentCount = 0;
8.     private void IncrementCount()
9.         => currentCount + +;
10. }
```

In this example, when the user clicks the **Increment** button, the event is sent to the server, where the **IncrementCount** method is executed. The updated **currentCount** value is then sent back to the client, and the UI is updated accordingly.

Lifecycle and state management

Blazor Server applications manage state on the server, which can lead to efficient use of resources and centralized control over application logic. The lifecycle of a Blazor Server component includes initialization, event handling, and state updates, all of which are managed by the server. For instance, consider a scenario where a user interacts with a form. The form data and state can be maintained on the server, ensuring consistent application behavior across sessions.

Following is a simplified example of handling a form submission:

```
1. @page "/submitform"
2. <h3>Submit Form</h3>
3.     <EditForm                                Model="userData"
   OnValidSubmit="HandleSubmit">
4.     <div>
5.         <label for="name">Name</label>
6.         <InputText id="name" @bind-Value="userData.Name" />
7.     </div>
8.     <div>
9.         <label for="email">Email</label>
10.        <InputText id="email" @bind-Value="userData.Email" />

```

```

>
11. </div>
12. <button type="submit">Submit</button>
13. </EditForm>
14.
15. @code {
16.     private User userData = new User();
17.
18.     private void HandleSubmit()
19.     {
20.         // Process form data on the server
21.     }
22.     private class User
23.     {
24.         public string Name { get; set; }
25.         public string Email { get; set; }
26.     }
27. }

```

In this example, when the form is submitted, the **HandleSubmit** method is called on the server, processing the user data securely and efficiently.

Performance and scalability

Blazor Server's server-side processing model can impact performance and scalability, especially under high user loads. For example, since all client interactions generate server requests, network latency and server processing time can become bottlenecks. Consider the following example where a data grid fetches data from a server-side database:

```

1. @page "/grid"
2. <h3>Data Grid</h3>
3. @if (data == null)
4. {
5.     <p>Loading...</p>
6. }
7. else
8. {

```



```

9.     <table>
10.         @foreach (var item in data)
11.         {
12.             <tr>
13.                 <td> @item.Id </td>
14.                 <td> @item.Name </td>
15.             </tr>
16.         }
17.     </table>
18. }
19. @code
20. {
21.     private List<Item> data;
22.     protected override async Task OnInitializedAsync()
23.     => data = await FetchDataFromDatabase();
24.     private Task<List<Item>> FetchDataFromDatabase()
25.     {
26.         return Task.FromResult(new List<Item> // Simulate
27.         {
28.             new Item { Id = 1, Name = "Item 1" },
29.             new Item { Id = 2, Name = "Item 2" },
30.         });
31.     }
32.     private class Item
33.     {
34.         public int Id { get; set; }
35.         public string Name { get; set; }
36.     }
37. }

```

In this case, each interaction potentially triggers server-side data retrieval and processing, which can strain server resources if not managed properly. Implementing features like data caching, efficient database queries, and proper server scaling strategies can help mitigate these issues.

Development experience

Blazor Server provides a rich development experience by allowing developers to use C# for both client-side and server-side logic

having a much more homogeneous codebase. Integration with tools like Visual Studio offers debugging, IntelliSense, and project templates, enhancing productivity. For example, debugging a Blazor Server application allows stepping through server-side code as it handles client events, providing insights into application behavior and state management.

Blazor WebAssembly

Blazor WebAssembly is a client-side framework within the Blazor ecosystem that allows developers to build interactive web applications using C# and .NET technologies. Unlike Blazor Server, Blazor WebAssembly applications run entirely in the user's browser. This is made possible through **WebAssembly (Wasm)** and Mono framework, a binary instruction format that enables high-performance execution of code on modern web browsers. Unfortunately, there is no direct compiler from C# to WebAssembly but we can execute the **intermediate language (IL)** directly in the browser thanks to the porting of Mono runtime to Wasm standard.

Following is a detailed exploration of Blazor WebAssembly, its architecture, and implementation:

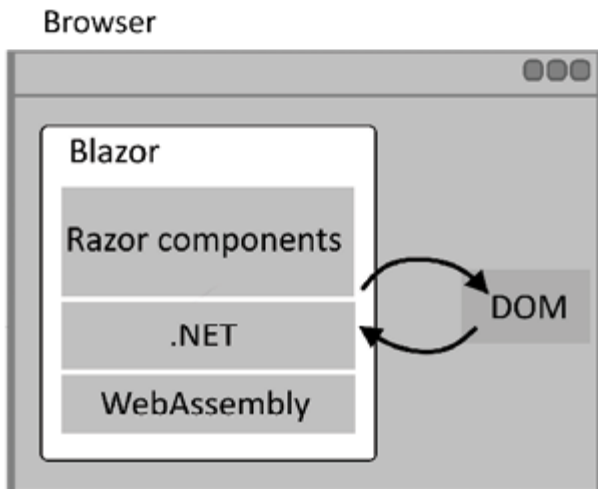


Figure 17.3: Blazor WebAssembly

What is WebAssembly

Wasm, is a low-level, binary instruction format designed for efficient execution and compact representation. It allows code written in various languages, including C#, C++, Rust, Go etc. to run on the web at near-native speed. It is designed to be a safe, portable, and efficient target for compilation of high-level languages, offering capabilities previously restricted to native applications. Runs in a secure sandbox environment, similar to JavaScript, but is not limited to the performance constraints of traditional interpreted languages. Browsers that support WebAssembly can execute Wasm modules alongside JavaScript, allowing developers to leverage existing web technologies.

Structure of WebAssembly

WebAssembly modules are composed of various sections that define the module's components. These sections include definitions for types, imports, functions, tables, memories, and global variables. Each function within a module has a unique identifier and can include parameters and local variables. The module also includes an export section for making functions, tables, memories, and globals accessible to the outside world. Additionally, a module may contain a start function that is executed upon instantiation.

A simple WebAssembly module structure might include sections for types, imports, functions, and exports.

Following is a textual representation of a module with a function that adds two numbers:

```
1. (module
2.   (type $add_type (func (param i32 i32) (result i32)))
3.   (func $add (type $add_type) (param $x i32) (param $y i32)
      (result i32)
4.     (i32.add
5.       (local.get $x)
6.       (local.get $y)))
7.   (export "add" (func $add))
8. )
```

This module defines a function **add** that takes two **i32** parameters and returns their sum. The function is exported under the name **add**.

Validation in WebAssembly

Validation in WebAssembly ensures that modules are well-formed and safe to execute. This process involves type-checking functions and their instruction sequences, ensuring that operations are performed on the correct types and that the stack is used correctly. Validation also checks for memory safety, such as ensuring that memory accesses are within bounds. The goal is to prevent runtime errors and security vulnerabilities by catching issues during the loading phase.

In the validation phase, Wasm ensures that all operations are type-safe. For example, the following invalid code would fail validation due to a type mismatch:

```
1. (module
2.   (func (param i32) (result i32)
3.     (f32.const 1.0) ;; Error: f32 on stack but expecting i32
4.     (i32.add (local.get 0) (i32.const 1))
5.   )
6. )
```

Here, attempting to add an **f32** value to an **i32** would cause a validation error.

Execution in WebAssembly

Execution in WebAssembly is carried out in two main phases: instantiation and invocation. During instantiation, a module's imports are resolved, and its components, such as memories and tables, are initialized. If a start function is defined, it is executed at this time. Invocation involves calling functions exported by the module, which can interact with the host environment through the defined imports and exports. The execution model follows a stack-based approach, where instructions manipulate values on an implicit operand stack.

After a module is validated, it can be instantiated and its functions called. For instance, if the module defined above is instantiated in a JavaScript environment, the **add** function can be called as follows:

```
1. const fs = require('fs');
2. const wasmBuffer = fs.readFileSync('add.wasm');
```

```
3. WebAssembly.instantiate(wasmBuffer).then(obj => {  
4.   console.log(obj.instance.exports.add(2, 3)); // Outputs: 5  
5. });
```

Binary format

The binary format of WebAssembly is a compact, efficient encoding that is designed for fast transmission and execution. It uses a stack-based virtual machine model, where instructions are encoded in a binary format that is smaller and faster to load than equivalent text representations. This format includes the module's structure, such as the sections for types, imports, functions, and more, encoded in a compact and structured manner.

For example, the above `add` function in binary format would begin with the following bytes:

```
00 61 73 6D 01 00 00 00 01 07 01 60 02 7F 7F 01 7F 03 02 01 00  
07 05 01 03 61 64 64 00 00 0A 09 01 07 00 20 00 20 01 6A 0B
```

These bytes include the Wasm magic number (`00 61 73 6D`), version (`01 00 00 00`), and the rest of the module's contents, including types, functions, and exports.

Text format

WebAssembly also defines a text format that is human-readable, making it easier to inspect and debug modules. This format uses a Lisp-like syntax to represent the module's structure and instructions. The text format serves as a more accessible way for developers to read and write Wasm code, facilitating understanding and troubleshooting.

For a more detailed exploration of WebAssembly's specifications, refer to the official WebAssembly Specification at <https://webassembly.github.io/spec/core/>.

Architecture of Blazor WebAssembly

In Blazor WebAssembly, the entire application, including the .NET runtime, is downloaded to the client's browser. This enables the application to execute entirely on the client-side without the need for server round-trips for processing events.

The architecture consists of the following key elements:

- **Client-side execution:** The application logic, UI components, and the .NET runtime all run within the browser. This means that after the initial download, interactions happen locally, providing a responsive user experience.
- **WebAssembly runtime:** The Blazor WebAssembly runtime loads the .NET assemblies and executes them using WebAssembly. The runtime is responsible for managing memory, handling exceptions, and executing C# code.
- **JavaScript interoperability:** While Blazor WebAssembly runs C# code, it can interoperate with JavaScript, allowing access to a wide range of web APIs and libraries not directly available in .NET.

Now, we can see what really is a component based Blazor WebAssembly application.

Razor component

Blazor apps are built using Razor components, which are elements of UI like pages, dialogs, or forms. These components are C# classes within .NET assemblies. Typically, components are written as Razor markup pages, which combine HTML and C# code to streamline UI development. The syntax, designed for productivity, is encapsulated in files with the **.razor** extension. While some refer to these as **Blazor components**, the official documentation consistently uses the terms **Razor components** or simply **components**.

In the previous section, we have seen a page for counter. We can use exactly the same code seen before for WebAssembly too. There is a single difference at runtime: in case of Blazor Server each time we click on the button a request to server will be sent, but in case of WebAssembly no requests are sent to server.

In the Counter page, we can see that the first line is **@page "/counter"**. This indicates that this piece of code should be treated as a page and subjected to routing rules. As a result, we can reach it directly by typing the address in our browser, for example, <https://localhost:5001/counter>. However, a standalone component cannot be addressed directly.

A standalone component in a web application cannot be used directly as a page because it is not associated with a specific URL through the routing system. On the other hand, a standalone component is usually a piece of UI that is used within a page or another component. It does not have its own routing definition and, therefore, cannot be directly accessed via a URL. For example, a component representing a button or a section of a form does not make sense as an independent entity accessible via a URL; it only makes sense within the context of a larger page that manages navigation and overall layout.

Let us see an example of component in Blazor that will be responsible to alternate the styles of rows in a table, as follows:

```
1. @typeparam TItem
2. <tr class="@RowClass">
3.     @foreach (var column in Columns)
4.     {
5.         <td>@column.Invoke(Item)</td>
6.     }
7. </tr>
8. @code {
9.     [Parameter] public TItem Item { get; set; }
10.    [Parameter] public int Index { get; set; }
11.    [Parameter] public List<Func<TItem, object>>
12.        Columns { get; set; }
13.    private string RowClass => Index % 2 == 0
14.        ? "table-row-even"
15.        : "table-row-odd";
16. }
```

This component will show alternate line background but it needs some input parameters. The data it needs are the following ones:

- **@typeparam TItem:** Allows the component to be generic and work with any data type.
- **Item:** The specific data representing the table row.
- **Index:** The index of the row, used to determine the style.
- **Columns:** A list of functions that extract the column values from Item.

As we can see from the code above, an input parameter must be decorated with **[Parameter]** attribute and must be a property. The use will be as follows:

```
1. <table class="table">
2. <thead>
3. <tr>
4. <th>Date</th>
5. <th>Temp. (C)</th>
6. <th>Temp. (F)</th>
7. <th>Summary</th>
8. </tr>
9. </thead>
10. <tbody>
11. @foreach (
12.     var (forecast, index)
13.     in forecasts.Select((forecast, index) => (forecast, index)))
14. {
15.     <TableRowComponent TItem="WeatherForecast"
16.         Item="forecast"
17.         Index="index"
18.         Columns="@((new List<Func<WeatherForecast,
19. object>> {
20.             x => x.Date.ToShortDateString(),
21.             x => x.TemperatureC,
22.             x => x.TemperatureF,
23.             x => x.Summary
24.         })" />
25.     }
26. </tbody>
27. </table>
```

The result will be as follows:

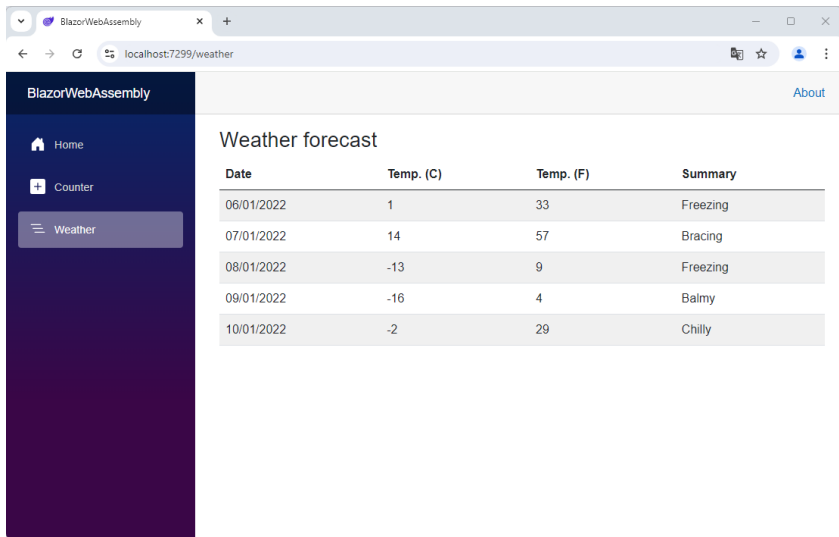


Figure 17.4: Component for alternate lines

As we can see, the rows styles are alternate.

Performance and optimization

In the rapidly evolving landscape of web applications, performance and optimization are critical factors that can make or break the user experience. Blazor WebAssembly, as part of .NET 9, presents exciting opportunities and unique challenges in this domain. The core principle of optimizing a Blazor WebAssembly application lies in minimizing the initial download size and optimizing runtime execution.

With .NET 8, several enhancements have been introduced to streamline the performance of Blazor WebAssembly applications. Techniques such as trimming unused assemblies and leveraging **ahead-of-time (AOT)** compilation have become more refined, significantly reducing payload size and improving load times. The introduction of features like improved JavaScript interop and better memory management further enhances runtime performance, making applications more responsive and efficient.

Looking ahead to .NET 9, promises even more advancements. The focus is on further reducing the size of the WebAssembly runtime and improving the performance of the rendering pipeline. Features such as enhanced lazy loading mechanisms and more sophisticated

data serialization formats are expected to play a crucial role in optimizing resource usage. Additionally, the .NET 9 includes experimental support for new compression algorithms and more granular control over caching strategies, allowing developers to fine-tune the delivery of their applications.

Implementing lazy loading of components and resources ensures that only the necessary parts of the application are loaded and rendered, reducing the burden on the client's browser. Effective use of caching strategies, such as service worker caching and proper HTTP caching headers, further enhances performance by minimizing redundant network requests.

Blazor Hybrid

Blazor Hybrid apps can be developed using various .NET native app frameworks, such as .NET MAUI, WPF, and Windows Forms. With BlazorWebView controls or WebView2 control, we can easily integrate Razor components into apps built with these frameworks. This integration makes it easy to create cross-platform Blazor Hybrid apps for both mobile and desktop using .NET MAUI, while also providing a means to modernize existing applications with WPF and Windows Forms.

These apps, being native, can utilize features beyond what is available on the web platform alone. They have full access to native platform capabilities through standard .NET APIs and can share and reuse components with Blazor Server or Blazor WebAssembly apps. Blazor Hybrid apps thus offer a unique combination of web, native app, and .NET platform advantages.

The hosting model for Blazor Hybrid apps allows for the reuse of existing components across different platforms, leveraging existing web development skills and resources. Moreover, these apps have complete access to the device's native capabilities. However, this model requires building, deploying, and maintaining separate native client apps for each platform. Additionally, users typically experience a longer process to find, download, and install these native apps compared to accessing a web app via a browser.

Blazor Hybrid with WPF

To build the application able to run Blazor code inside a WPF application, for the first step we have to create a standard WPF application. After created a default WPF application, we have to add the **nuget** package **Microsoft.AspNetCore.Components.WebView.Wpf** then we have to apply the following modifications to the **.csproj** file directly. After opening the **.csproj** file, on the top replace the first line with the following:

```
1. <Project Sdk="Microsoft.NET.Sdk.Razor">
```

Then in the first **<PropertyGroup>** we have to add the following line:

```
1. <RootNamespace>WpfBlazor</RootNamespace>
```

At this point, we are ready to start adding the necessary files to transform our WPF app into a real hybrid application by following these steps:

1. **Add _Imports.razor:** in the root of our project, add file named **_Imports.razor** and in this file add **@using Microsoft.AspNetCore.Components.Web**.
2. **Add wwwroot folder:** always in the root of our project, add a folder named **wwwroot** and inside add a file named **index.html** and a folder named **css** where we will add a file **app.css** and another folder named **bootstrap** as follows:

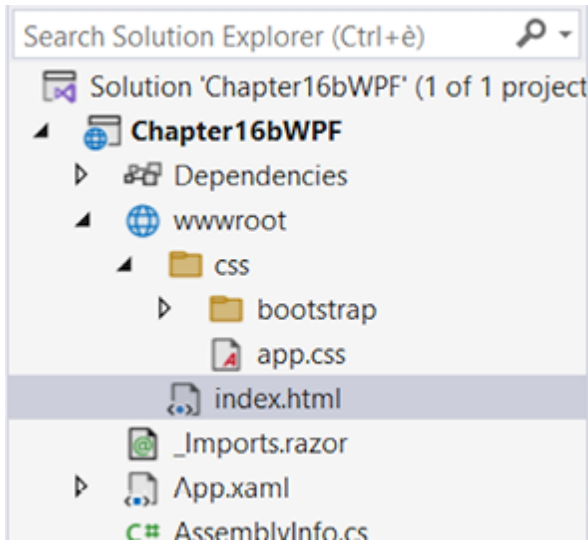


Figure 17.5: Blazor WPF hybrid file structure

3. **Add file contents:** Now we have to fill the files created in previous point, as follows:

o **Index.html:**

```

1. <!DOCTYPE html>
2. <html lang="en">
3.
4. <head>
5.   <meta charset="utf-8" />
6.   <meta name="viewport"
   content="width=device-width, initial-scale=1.0" /
   >
7.   <title>WpfBlazor</title>
8.   <base href="/" />
9.   <link href="css/bootstrap/bootstrap.min.css"
   rel="stylesheet" />
10.  <link href="css/app.css" rel="stylesheet" />
11.  <link href="WpfBlazor.styles.css"
   rel="stylesheet" />
12. </head>
13.
14. <body>

```

```

15. <div id="app"> Loading... </div>
16.
17. <div id="blazor-error-ui" data-nosnippet>
18.     An unhandled error has occurred.
19.     <a href="" class="reload"> Reload </a>
20.     <a class="dismiss"> ✕ </a>
21. </div>
22. <script src="_framework/
    blazor.webview.js"> </script>
23. </body>
24.
25. </html>

```

o App.css:

```

1. html, body {
2.     font-family: 'Helvetica Neue', Helvetica, Arial, sans-
        serif;
3. }
4.
5. h1:focus {
6.     outline: none;
7. }
8.
9. a, .btn-link {
10.     color: #0071c1;
11. }
12.
13. .btn-primary {
14.     color: #fff;
15.     background-color: #1b6ec2;
16.     border-color: #1861ac;
17. }
18.
19. .valid.modified:not([type=checkbox]) {
20.     outline: 1px solid #26b050;
21. }

```

```

22.
23. .invalid {
24.     outline: 1px solid red;
25. }
26.
27. .validation-message {
28.     color: red;
29. }
30.
31. #blazor-error-ui {
32.     background: lightyellow;
33.     bottom: 0;
34.     box-shadow: 0 -1px 2px rgba(0, 0, 0, 0.2);
35.     display: none;
36.     left: 0;
37.     padding: 0.6rem 1.25rem 0.7rem 1.25rem;
38.     position: fixed;
39.     width: 100%;
40.     z-index: 1000;
41. }
42.
43. #blazor-error-ui .dismiss {
44.     cursor: pointer;
45.     position: absolute;
46.     right: 0.75rem;
47.     top: 0.5rem;
48. }

```

Now we are ready to download Bootstrap from the official website (<https://getbootstrap.com/> in Docs section then Download). We have to download the latest version of file **bootstrap.min.css** and put it into the **wwwroot/css/bootstrap** folder created before.

4. **Add razor file:** only for example, we can add the classic Counter file as usual in Blazor template. Add the **counter.razor** file in the root folder of our project with the following content:

```

1. <h1>Counter</h1>
2. <p>Current count: @currentCount</p>
3.     <button class="btn btn-primary"
        @onclick="IncrementCount">Click me</button>
4. @code {
5.     private int currentCount = 0;
6.     private void IncrementCount()
7.     => currentCount++;
8. }

```

5. **Modify MainWindow.xaml:** add the following contents to the **MainWindow.xaml** so that we can recall the proper page created in previous point. First of all, add the namespace as follows:

xmlns:blazor="clr-

namespace:Microsoft.AspNetCore.Components.WebView.Wpf;assembly=

Then, add the grid content as follows:

```

1. <Grid>
2.     <blazor:BlazorWebView HostPage="wwwroot
        \index.html" Services="{DynamicResource services}">
3.         <blazor:BlazorWebView.RootComponents>
4.             <blazor:RootComponent Selector="#app"
                ComponentType="{x:Type local:Counter}" />
5.         </blazor:BlazorWebView.RootComponents>
6.     </blazor:BlazorWebView>
7. </Grid>

```

Now, in the code behind of the **MainWindow.xaml** add the following code after the **InitializeComponent** method call:

```

1. var serviceCollection = new ServiceCollection();
2. serviceCollection.AddWpfBlazorWebView();
3. Resources.Add("services",
    serviceCollection.BuildServiceProvider());

```

Now, we are ready to run our WPF hybrid application. The output will be as follows:

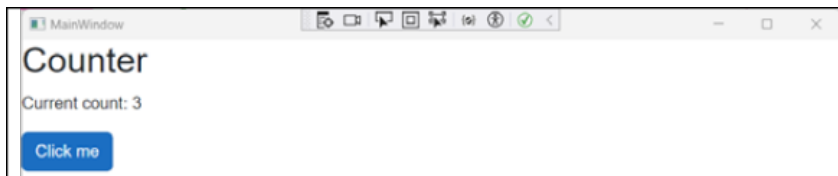


Figure 17.6: Blazor WPF hybrid application running

As we can see, the razor page counter is shown in our WPF application.

Blazor Hybrid with Windows Forms

The Windows Forms hybrid application, is not so far as WPF hybrid application. All the points are almost the same but the **nuget** package and some other little details.

The **nuget** package to add will be **Microsoft.AspNetCore.Components.WebView.WindowsForms** then we have to apply the same modifications to the **.csproj** file as we made in WPF application. Then we have to add the same files and folders.

At the end when our project will be ready, we have to modify the **Form1.cs** file adding from the toolbox the **BlazorWebView** control. Set the property **Dock** of the control **BlazorWebView** to **fill** so that the control will fill the whole form.

Now, the last modification to the code behind **Form1**. Add the following code to the constructor after the **InitializeComponent** method call:

1. `var services = new ServiceCollection();`
2. `services.AddWindowsFormsBlazorWebView();`
3. `blazorWebView1.HostPage = "wwwroot\\index.html";`
4. `blazorWebView1.Services = services.BuildServiceProvider();`
5. `blazorWebView1.RootComponents.Add<Counter>("#app");`

Now, we can run the application. We will see a form as follows:



Figure 17.7: Blazor Windows Forms hybrid application running

As we can see, the razor page counter is shown in our Windows Forms application.

Blazor Hybrid with MAUI

Blazor hybrid with MAUI offers a distinct approach compared to traditional hybrid applications built with WPF and Windows Forms. While WPF and Windows Forms are primarily focused on desktop application development for Windows, this framework expands the horizon by providing a unified platform for building cross-platform applications that run on Android, iOS, macOS, and Windows.

With Blazor integrated, it allows the use of web technologies like HTML, CSS, and Razor components within native applications. This combination leverages the strengths of both Blazor for web development and MAUI for native app performance and access to platform-specific features.

This hybrid approach enables the creation of rich, interactive user interfaces with the same codebase across multiple platforms, enhancing productivity and reducing maintenance efforts. The model not only modernizes the development process but also ensures a seamless and consistent user experience across different devices and operating systems.

.NET MAUI overview

MAUI is the acronym of Multi-platform App UI and is a recent framework able to create native mobile and desktop applications through C# and XAML. Now, it is possible to use Blazor as front-end language instead of XAML.

.NET MAUI is an open-source framework and the evolution of **Xamarin.Forms**. It expands beyond mobile development to include desktop scenarios, with UI controls that have been rebuilt from the

ground up for better performance and extensibility.

If we have previously used **Xamarin.Forms** to create cross-platform user interfaces, we will find many similarities in .NET MAUI, though there are notable differences. With .NET MAUI, we can develop multi-platform applications within a single project, while still having the flexibility to add platform-specific source code and resources when necessary.

One of the main goals of .NET MAUI is to enable us to implement most of our app logic and UI layout within a unified codebase, simplifying the development process and improving efficiency.

.NET MAUI with Blazor

.NET MAUI has by default the BlazorWebView control then we do not have to add any packages. We have to only create a MAUI project with the template **.NET MAUI Blazor Hybrid App** as follows:

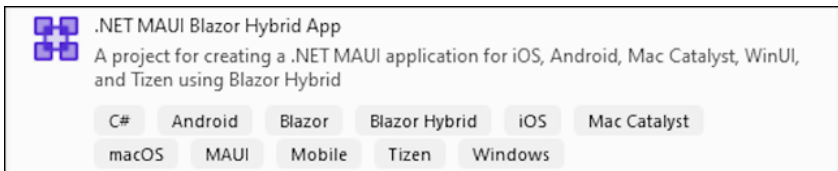


Figure 17.8: .NET MAUI project template

This template will provide to configure the solution and the project with all the necessary configurations. We will find all the razor pages, the **wwwroot** folder with all the necessary files, a Resources folder where there will be all the resources files. A file **MainPage.xaml** it is placed in the root of the project as entry point and main page for the application. In this page we can find the **BlazorWebView** component able to show our Blazor pages as follows:

1. `<BlazorWebView x:Name="blazorWebView"`
2. `HostPage="wwwroot/index.html">`
3. `<BlazorWebView.RootComponents>`
4. `<RootComponent Selector="#app"`
5. `ComponentType="`
6. `{x:Type local:Components.Routes}"`
7. `</BlazorWebView.RootComponents>`

8. </BlazorWebView>

At this point everything is ready for a first run. The application shows the standard Blazor template application with a single particularity that there is no connection to internet and will shows as follows:

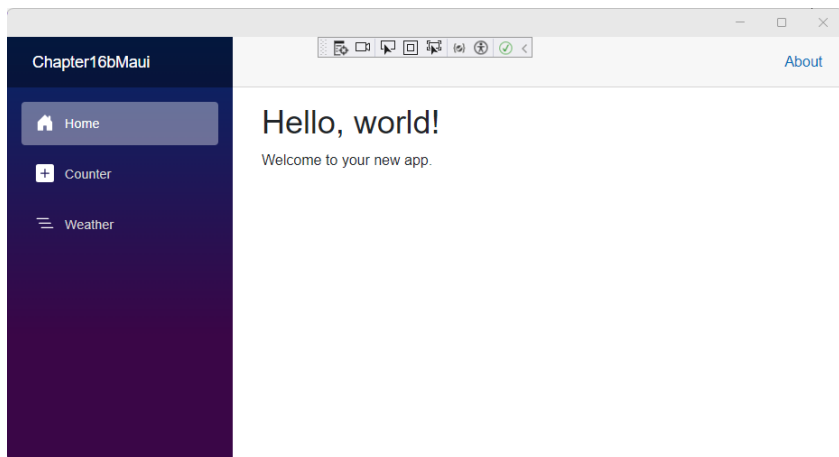


Figure 17.9: .NET MAUI application

In summary, MAUI with Blazor represents a powerful combination that enables developers to create cross-platform applications using C# and HTML/CSS, leveraging both the native capabilities of .NET MAUI and the flexibility and familiarity of Blazor for web development. This synergy not only accelerates the development process but also ensures a consistent and high-quality user experience across different platforms.

Blazor render modes

.NET 8 introduced a revolutionary level of flexibility in application hosting models, allowing developers to break free from the limitations of a single hosting approach. This marked a new era of modularity, enabling developers to address diverse needs. In .NET 9, Blazor enhances flexibility, allowing precise customization of page and component rendering for tailored experiences and optimized performance.

Rather than being confined to a single hosting model for an entire

application, Blazor now allows developers to mix and match hosting models within the same app. For example, a static contact page might use static rendering or server-side rendering for faster load times and better SEO. In contrast, a real-time dashboard might employ WebAssembly to take advantage of client-side processing capabilities, enabling smoother and more dynamic interactions.

This flexibility is made possible by render modes, which empower us to make the most suitable architectural choices for each scenario. Render modes are categorized as follows:

- **Static:** for rendering non-interactive HTML content
- **InteractiveServer:** for components rendered server-side, with interactivity enabled via SignalR
- **InteractiveWebAssembly:** For components rendered entirely on the client using WebAssembly

In practice, we have to create a new project with Blazor Web App template as follows:

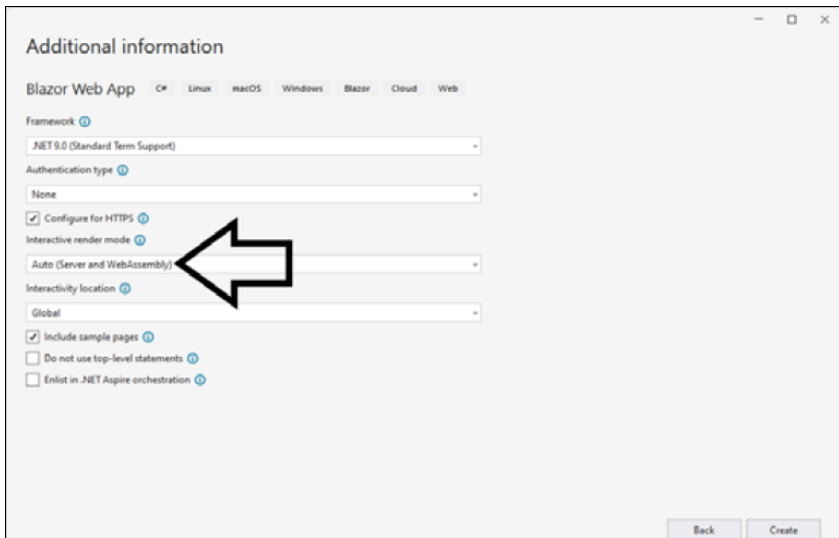


Figure 17.10: Blazor Web App

This solution will generate two projects: the first is the **Server project** and the second with the **.Client** suffix in the project name that will be involved in client-side rendering using WebAssembly. In this solution we can find a **Counter.razor** page as usual but with a

notable difference: the inclusion of the **@rendermode** directive, as follows:

1. @rendermode InteractiveAuto

The **@rendermode** directive specifies how the page is rendered. The available options are the ones mentioned above: **InteractiveAuto**, **InteractiveServer** and **InteractiveWebAssembly**.

Blazor InteractiveServer

Server-side rendering in Blazor refers to the process where the server generates the initial HTML content for a web application and sends it to the client. This technique is primarily used with the Blazor Server hosting model, where the app's components are executed on the server, and interactions between the client and server are managed through a SignalR connection. This architecture enables a responsive and interactive user interface by delegating most of the application logic and lifecycle management to the server.

SSR in Blazor offers a faster initial load time since the HTML is rendered and sent from the server, which can be particularly beneficial for **search engine optimization (SEO)** as search engines can more easily index server-rendered content. This approach also means that the client device does not need to handle heavy processing tasks, making it suitable for less powerful devices. Prerendering involves initially rendering the page content on the server without enabling event handlers for the rendered controls. This process sends the HTML UI of the page quickly in response to the initial request, enhancing the app's perceived responsiveness. Prerendering also benefits SEO by providing content in the initial HTTP response that search engines can use to calculate page rank. This process is always followed by final rendering, which can occur either on the server or the client.

Blazor InteractiveWebAssembly

Blazor client-side rendering, is an approach to building web user interfaces using C# and .NET, where all the application code runs directly in the browser. This is enabled by WebAssembly, and in the particular case of Blazor full stack web UI with the directive

@rendermode **InteractiveWebAssembly** as seen previously.

With **InteractiveWebAssembly**, the component or page, including all necessary resources such as components, styles, and business logic, is downloaded to the user's browser. This allows the application to function independently of the server once loaded, even supporting offline operations. User interactions are handled quickly and smoothly because operations are executed locally in the browser without the need for constant communication with the server.

However, this approach also presents some challenges, such as a potentially **slower initial load time** since the entire application code must be transferred from the server to the client. Additionally, since the application runs within the browser's environment, it is limited by the client's capabilities, including available memory and processor performance.

Blazor InteractiveAuto

InteractiveAuto is a rendering mode in Blazor that allows the framework to select the most appropriate rendering strategy for a component or page at initialization, based on the client's capabilities and the application's configuration. This approach enables the use of **InteractiveServer** or **InteractiveWebAssembly**, depending on the runtime context. When the client does not support WebAssembly or has limited resources, Blazor defaults to **InteractiveServer**, rendering components on the server and delivering them as static HTML to the client while maintaining interactivity through a SignalR connection. This strategy ensures faster initial load times and supports SEO by providing server-rendered content that search engines can index effectively. If the client supports WebAssembly, **InteractiveWebAssembly** enables the application to run entirely in the browser, ideal for rich interfaces with smooth interactions and reduced server load. **InteractiveAuto** optimizes performance and scalability by selecting the best rendering mode at runtime, which remains fixed during the page or component lifecycle.

Blazor StreamRendering attribute

Imagine we have a page in an application that needs to fetch real-

time data asynchronously from a database or an external API. In traditional SSR, the server waits for all asynchronous operations to complete before sending the full HTML content to the client, potentially delaying the page load.

In cases where the page does not require additional interactivity, streaming rendering offers a simpler and more efficient alternative to fully interactive solutions like `InteractiveServer` or `InteractiveWebAssembly`. This is where streaming rendering offers an efficient alternative.

With **StreamRendering** in .NET9, the server generates the initial HTML for the page and includes placeholders for any data that will be fetched asynchronously. This partial HTML is sent to the client immediately, allowing the browser to start rendering the page without delay. Meanwhile, the server keeps the connection open. As the asynchronous data becomes available, the server sends the remaining HTML content through the same open connection. We can switch on this function with the following directive:

1. `@page "/weather"`
2. `@attribute [StreamRendering]`

On the client side, Blazor dynamically replaces the placeholder content with the newly received data, seamlessly updating the page. This technique enhances the user experience by significantly improving perceived load times, as users can interact with the initial content while waiting for the remaining data to load. Additionally, **StreamRendering** is ideal for scenarios where interactivity is not needed, but progressive data loading is beneficial, such as displaying reports or live feeds.

Blazor United

Blazor United is the term originally used by the community to describe the evolution of Blazor into a unified model that integrates multiple rendering approaches within a single framework. This term describes Blazor's ability to combine client-side and server-side rendering to enable the development of modern web applications with a consistent and integrated experience. This approach facilitates the creation of smooth and interactive user experiences while maintaining optimal resource management and a unified

codebase.

As of .NET 9, this feature is fully integrated into Blazor, leveraging the new render modes such as `InteractiveServer` and `InteractiveWebAssembly`.

Conclusion

This chapter provided an overview of Blazor, emphasizing its flexibility with both Blazor Server and Blazor WebAssembly hosting models. These options allow applications to run server-side or entirely in the user's browser, catering to various scenarios. Blazor's component-based architecture and ability to leverage existing .NET skills simplify full-stack development without the need for JavaScript, providing a consistent programming model across client and server. With support for dependency injection, data binding, and other modern web features, Blazor is well-suited for building scalable, maintainable applications. As a cross-platform framework, it empowers developers to deliver rich, interactive web experiences on any device.

Now, that we have a real comprehension of what Blazor is, in the next chapter, we will explore Blazor more thoroughly, focusing on deeper aspects and understanding the complexities and capabilities of the framework beyond the basics.

Points to remember

- **Component-based:** Blazor apps are built with reusable components.
- **Data binding:** Supports both one-way and two-way data binding.
- **Interactive UIs:** Blazor enables interactive web user interfaces.
- **Cross-platform:** Blazor works on any platform that supports WebAssembly.

Multiple choice questions

1. What language is primarily used for writing Blazor

applications?

- a. JavaScript
- b. Python
- c. C#
- d. Java

2. Which of the following is a Blazor hosting model?

- a. Blazor Mobile
- b. Blazor Server
- c. Blazor API
- d. Blazor Cloud

3. In Blazor, what does the [Parameter] attribute denote?

- a. A local variable
- b. A public property
- c. A component parameter
- d. A method

4. What is the purpose of the @page directive in Blazor?

- a. To specify a page's route
- b. To declare a component
- c. To import namespaces
- d. To set layout

5. Which library is commonly used for real-time communication in Blazor Server apps?

- a. SignalR
- b. jQuery
- c. Angular
- d. React

6. How does Blazor handle data binding?

- a. Only one-way binding
- b. Only two-way binding

- c. One-way and two-way binding
 - d. No data binding
7. What is the purpose of the `@code` directive in Blazor components?
- a. To define C# code in a component
 - b. To include CSS
 - c. To import JavaScript files
 - d. To set the component's layout
8. What does the `[Inject]` attribute in Blazor do?
- a. Marks a method for logging
 - b. Inject an object from dependency injection in C#
 - c. Handles user authentication
 - d. Defines a route

Answer key

Key terms

- **Blazor:** Microsoft framework for building interactive web applications using C# and .NET instead of JavaScript.
- **Blazor server:** is a hosting model for Blazor where the app's execution happens on the server, with UI updates via SignalR.
- **Blazor WebAssembly:** is a hosting model where the app runs directly in the user's browser using WebAssembly, enabling full client-side functionality.
- **Interactive render modes:** determine how UI components are rendered, including Server, WebAssembly, and hybrid modes.
- **Blazor stream reading:** The `[StreamReading]` attribute in Blazor allows for efficient streaming of large file uploads directly to a stream without buffering in memory.

- **Command pattern:** Decouple the requester of an operation from the object that performs the action.

CHAPTER 18

Advanced User Interfaces with Blazor

Introduction

In this chapter, we will explore how to make Blazor applications more functional and interactive. We will begin by validating user inputs to ensure data accuracy and compliance; we will learn how to pass information smoothly between pages and components without losing data; we will also discover how to integrate Blazor with JavaScript, enhancing our ability to use browser functions. Finally, we will focus on managing the application state to preserve data on the front end. Together, these skills will help us create dynamic, responsive user experiences.

Structure

In this chapter, we will cover the following topics:

- Input form validation
- Passing parameters through application pages
- JavaScript interop

- Basic state management

Objectives

By the end of this chapter, readers will be equipped with the skills to manage Blazor professionally by ensuring accurate data entry and validation, seamless parameter transfer, and effective state management. We will also explore interaction between Blazor and JavaScript through interoperability to enhance functionalities, improving interactivity and the overall user experience in web projects.

Input form validation

An input form refers to a form component used for handling user input and managing user input submissions. Blazor offers various features and components that make creating forms and handling user data straightforward and efficient.

Following is an example of a Blazor component containing a form:

```
1.      <EditForm                                Model="@person"
OnValidSubmit="HandleValidSubmit">
2.      <DataAnnotationsValidator />
3.      <ValidationSummary />
4.
5.      <div>
6.          <label for="name">Name: </label>
7.          <InputText id="name" @bind-Value="person.Name" />
8.          <ValidationMessage For="@(() => person.Name)" />
9.      </div>
10.
11.     <div>
12.         <label for="surname">Surname: </label>
13.         <InputNumber id="surname" @bind-
Value="person.Surname" />
14.         <ValidationMessage For="@(() => person.Surname)" /
>
15.     </div>
16.
```

```
17.     <button type="submit"> Submit </button>
18. </EditForm>
19.
20. @code {
21.     private Person person = new Person();
22.
23.     private void HandleValidSubmit()
24.     {
25.         // Handle the form submission
26.     }
27. }
```

An input form in Blazor is built using the `<EditForm>` component. This component creates and manage the input of users. It binds the form fields to a model, allowing for two-way data binding. This means that any changes made by the user in the form fields will automatically update the corresponding properties in the model, and any changes in the model will reflect in the form fields.

Form validation

Validation is an important aspect of form handling, ensuring that the data users enter meets specific criteria before it is processed or saved. Blazor provides a robust validation system that integrates seamlessly with its form components, making it easy to enforce and display validation rules. Blazor's validation system is built around data annotations and custom validation logic. These validation rules are typically applied to the model class properties bound to form fields. When a user submits a form, Blazor checks the input against these rules and provides feedback if the data does not meet the required criteria.

Data annotations

Data annotations are attributes that we can add to the properties of a model class to specify validation rules.

Commonly used data annotations are as follows:

- **[Required]**: Ensures that a field is not left empty.
- **[Range(min, max)]**: Validates that a number falls within a specified range.

- **[StringLength(max)]**: Limits the length of a string.
- **[RegularExpression(pattern)]**: Validates that the input matches a specific pattern, such as an email address format.
- **[Compare]**: Compares the value of one property with another.
- **[EmailAddress]**: Validates that a property is a well-formed email address.
- **[Phone]**: Validates that a property is a valid phone number.
- **[CreditCard]**: Validates that a property is a valid credit card number.
- **[Url]**: Ensures that a property is a valid URL.
- **[CustomValidation]**: Allows for custom validation logic to be implemented.
- **[MaxLength]**: Specifies the maximum length of a string or array.
- **[MinLength]**: Specifies the minimum length of a string or array.
- **[FileExtensions]**: Validates the file extension of some files.
- **[DataType]**: Specifies the type of data, such as Date, Time, PhoneNumber, etc.
- **[Display]**: Specifies how a property should be displayed.

Note: While not strictly a validation attribute, it is often used alongside validation attributes to improve UI.

- **[Timestamp]**: Ensures that a property is treated as a timestamp in a concurrency check.

These data annotations can be combined to create comprehensive validation rules for forms in Blazor applications. They help ensure that the data collected meets the business requirements before being processed or saved.

Following is an example:

1. `public class Person`
2. `{`
3. `[Required(ErrorMessage = "Name is required.")]`

```

4.     public string Name { get; set; }
5.     [Required(ErrorMessage = "Surname is required.")]
6.     public string Surname { get; set; }
7.     [Range(18, 100, ErrorMessage = "Age must be between 18
and 100.")]
8.     public int Age { get; set; }
9. }

```

In this example, the **Name** property must not be empty, and the **Age** property must be between 18 and 100. If these rules are not met, Blazor will automatically display the associated error messages.

Implementing validation in Blazor

As we have seen in previous section, to use validation we have to wrap the form fields in an **<EditForm>** component, and specify the model to which the form fields are bound. Inside the **<EditForm>**, we can use input components like **<InputText>** or **<InputNumber>**, which are specialized Blazor components designed for form handling.

The components available for form input in Blazor includes the following:

- **<InputText>**: Used for basic text input fields.
- **<InputTextArea>**: Used for multi-line text input (textarea).
- **<InputNumber>**: Used for numeric input (integer or decimal).
- **<InputDate>**: Used for date input.
- **<InputCheckbox>**: Used for Boolean input.
- **<InputSelect>**: Used for dropdown selection.
- **<InputFile>**: Used for file upload input.
- **<InputRadioGroup>**: Used to group multiple **<InputRadio>** buttons.
- **<InputRadio>**: Used for radio button input. Used inside an **<InputRadioGroup>**.

In addition to the previous elements, we can also have all HTML tags.

Designing a form in Blazor requires that the data entered meets

some key criteria to ensure the correctness and reliability of the application.

To guide this process, it is necessary to follow three principles as follows:

- Validation activation
- Validation feedback
- Model binding

Validation activation is an important part of ensuring the accuracy of the data entered. The `<DataAnnotationsValidator/>` component ensures that the validation system based on the data annotations defined in the **Person** class, is properly activated. This setup, allows for automatic checking of the input data against the specified rules, ensuring that any mistakes or omissions are promptly flagged.

Providing users with clear and immediate feedback is another important aspect of form handling in Blazor. **Validation feedback** is handled through two primary components: `<ValidationSummary/>` and `<ValidationMessage>`. The `<ValidationSummary/>` component is used to display a comprehensive list of all validation errors, giving the user an overview of any issues that need to be addressed. On other hand, `<ValidationMessage>` components are more targeted, showing specific error messages related to individual fields, such as the **Name**, **Surname** or **Age** fields in the form through the parameter **For** that accept a function to get the specific validation. This specific feedback helps users quickly and effectively correct their input. The core of this system is **model binding**, where the form is directly tied to a specific object, such as a **person**. By using the **Model="@person"** attribute for the `<EditForm>` component, each field in the form are linked to the corresponding property within the **person** object. This connection ensures that any changes made in the form are immediately reflected in the object's properties.

Following is the complete example of how this looks in a Blazor form:

1. `@page "/person"`
2. `@using System.ComponentModel.DataAnnotations`
3. `@rendermode InteractiveAuto`

```

4.         < EditForm                                     Model = "@person"
OnValidSubmit = "HandleValidSubmit" >
5.     < DataAnnotationsValidator />
6.     < ValidationSummary />
7.     < div >
8.         < label for = "name" > Name: < /label >
9.         < InputText id = "name" @bind-Value = "person.Name" />
10.        < ValidationMessage For = "@(() => person.Name)" />
11.    < /div >
12.    < div >
13.        < label for = "name" > Surname: < /label >
14.        < InputText id = "name" @bind-Value = "person.Surname" /
>
15.        < ValidationMessage For = "@(() => person.Surname)" /
>
16.    < /div >
17.    < div >
18.        < label for = "age" > Age: < /label >
19.        < InputNumber id = "age" @bind-Value = "person.Age" />
20.        < ValidationMessage For = "@(() => person.Age)" />
21.    < /div >
22.    < button type = "submit" > Submit < /button >
23. < /EditForm >
24. @code {
25.     private Person person = new Person();
26.     private void HandleValidSubmit()
27.     {
28.         // Logic for handling form submission when the input is
valid
29.     }
30. }

```

In case we use only the `<DataAnnotationsValidator/>` and `<ValidationSummary/>` the aspect of our form will be as follows:

Chapter18Form

About

- Surname is required.
- Age must be between 18 and 100.

Name: Dario

Surname:

Age: 0

Submit

Figure 18.1: Blazor validation form summary

In case we want to use only the `<DataAnnotationsValidator/>` and `<ValidationMessage/>` for each field, the output screen will be as follows:

Chapter18Form

About

Name: Dario

Surname:

Surname is required.

Age: 0

Age must be between 18 and 100.

Submit

Figure 18.2: Blazor validation form for each field

Together, these elements (validation feedback, validation activation and model binding) work in harmony to create a robust form-handling experience in Blazor applications, ensuring that data entry is both user-friendly and reliable.

Custom validation

While data annotations cover many common scenarios, sometimes we need more complex validation logic. In such cases, we can implement custom validation by creating a class that inherits from

ValidationAttribute and overriding the IsValid method.

Following is an example:

```
1. public class CustomAgeValidation : ValidationAttribute
2. {
3.     protected override ValidationResult IsValid
4.         (object value, ValidationContext validationContext)
5.     {
6.         int age = (int)value;
7.         if (age < 18)
8.         {
9.             return new ValidationResult
10.                ("Age must be 18 or older.");
11.         }
12.         return ValidationResult.Success;
13.     }
14. }
```

In this example, **CustomAgeValidation** enforces that the age must be at least 18. We can apply this custom validation to any property just like a standard data annotation. In this case, we can localize the message, adding more complex validation rules and the range can be dynamic by configuration.

Passing parameters through application pages

In modern web applications, the ability to pass data through application pages or components is one of the necessary features for creating dynamic content and enabling user interaction. Whether we are navigating from one page to another, understanding how to effectively pass data and information between contexts is very important.

The structure of any application involves a hierarchy among its various components, with a parent entity and a child entity. Data transfer between these contexts can occur in two directions, that is, from the parent to the child or from the child to the parent. In Blazor context, the technique to transfer data from parent to child is named **Parameter** and from child to parent is **EventCallback<T>**. This bidirectional communication ensures that different parts of the

application can interact and respond to each other effectively, maintaining a cohesive and dynamic system where changes in one context can influence the behavior or state of another.

Parameters

Parameters in Blazor serve as a mechanism to transmit data from one context to another, ensuring that the receiving entity can access and utilize the provided information.

In a Blazor context, we often work with parameters, which allow seamless data transfer between different layers or entities. These parameters enable dynamic behavior and customization by providing the necessary information to the receiving context, ensuring that each part can function cohesively within the broader application framework.

In the context of a Blazor application, we can have three main parameters type as follows:

- Pages parameters
- Components parameters
- Cascading parameters

Let us see what they are.

Pages parameters

In the context of pages, we can have different type of parameters as follows:

Route parameters

We can define route parameters by specifying them in the `@page` directive of Blazor page. For example, we can imagine have a product page able to show the data of a single product from his `id` as follows:

1. `@page "/product/{id}"`
- 2.
3. `@code {`
4. `[Parameter]`
5. `public int Id { get; set; }`
- 6.

```
7. protected override void OnParametersSet()
8. {
9.     // Logic
10. }
11. }
```

In this example, **id** is a route parameter. When the user navigates to **/product/123**, the **Id** property in the page component will be set to **123**.

Optional parameters

We can also define optional parameters in a Blazor page by using **?** in the route template.

Following is an example:

```
1. @page "/product/{id:int?}"
2. @code {
3.     [Parameter]
4.     public int? Id { get; set; }
5. }
```

Here, **id** can be **null**, meaning the URL might be **/product/** or **/product/123**, and the component will handle both cases.

Multiple parameters

We can also define multiple route parameters in a Blazor page as follows:

```
1. @page "/product/{category}/{id:int}"
2. @code {
3.     [Parameter]
4.     public string Category { get; set; }
5.     [Parameter]
6.     public int Id { get; set; }
7. }
```

In this case, both **Category** and **Id** are parameters passed through the URL, such as **/product/electronics/123**.

Query strings

A special mention is for Query strings as pages parameters behavior. Query strings are one of the most commonly used methods for passing parameters between pages. They are appended to the URL

and can be accessed by the target page. For example, if we have a URL as follows:

[https://example.com/products?
category=electronics&sort=price](https://example.com/products?category=electronics&sort=price)

Category and **sort** are the parameters passed to the **products** page. This method is straightforward to implement and widely supported across different web frameworks. The primary advantage of using query strings is their simplicity; it allows us to easily share parameters through URLs or bookmarks without requiring server-side processing to extract the parameters. However, this approach also has some limitations and several pitfalls. Since query strings are visible in the URL, they are less secure for transmitting sensitive data. Additionally, the length of query strings is limited, which can be restrictive when dealing with complex data. Users can also modify query strings, leading to potential security risks if we do not properly validate the inputs.

If we want to use query string parameters instead of route parameters, we can use the **Microsoft.AspNetCore.WebUtilities.QueryHelpers** class with **NavigationManager** to parse the query string manually within our component. However, query string parameters are not automatically mapped to **[Parameter]** properties in the same way route parameters are.

If we decorate a **[Parameter]** property with the **[SupplyParameterFromQuery]** attribute (e.g., **[SupplyParameterFromQuery(Name = "id")]**), the framework automatically binds this property to the value of the id query parameter in the URL, if present.

Following is an example on how to treat **query** string in a Blazor page:

1. `@inject NavigationManager Navigation`
2. `@code {`
3. `private string Category;`
4. `private int? Id;`
5. `protected override void OnInitialized()`
6. `{`
7. `var uri = Navigation.ToAbsoluteUri(Navigation.Uri);`

```

8.     var query = System.Web.HttpUtility.ParseQueryString(uri.
        Query);
9.     Category = query["category"];
10.    if (int.TryParse(query["id"], out var id))
11.        Id = id;
12.    }
13. }

```

Here, the **OnInitialized** method parses the query string manually to extract parameters and assign the values to the correspondent properties.

Components parameters

Components parameters are defined using the **[Parameter]** attribute applied to public properties within the child component. This allows the parent component to set values that the child component can use to render content or perform logic.

For example, consider a scenario where we have a parent component that wants to pass a string value to a child component for display. In the child component, we have to define a property as follows:

```

1. @code {
2.     [Parameter] public string Message { get; set; }
3. }

```

In the parent component, we would then pass a value to this parameter when including the child component as follows:

```

1. <ChildComponent Message="Hello, Chapter 18!" />

```

Here, **Message** is a parameter in the **ChildComponent** that gets its value from the parent component. The child component can now use this **Message** property to display the past value or perform any other required logic. By using the **[Parameter]** attribute alongside **[EditorRequired]**, we can ensure that a parameter is required and must be provided by the parent component. Parameters in Blazor also support **two-way** data binding simply using **@bind="property_name"** as parameter. The **@bind** directive under the hood create two properties: the main property where store the value and the **EventCallback** property able to return back the value. This knowledge is fundamental for multiple binding on a same

component. For example, we can imagine a single component able to manage two features. The first will be the dark mode and the second the visibility of a panel as follows:

```
1. private bool isDarkMode;
2. [Parameter]
3. public bool IsDarkMode
4. {
5.     get => isDarkMode;
6.     set
7.     {
8.         isDarkMode = value;
9.         _ = IsDarkModeChanged.InvokeAsync(isDarkMode);
10.    }
11. }
12. [Parameter]
13. public EventCallback<bool> IsDarkModeChanged { get; set; }
14.
15. private bool isVisible
16. [Parameter]
17. public bool IsVisible
18. {
19.     get => isVisible;
20.     set
21.     {
22.         isVisible = value;
23.         _ = IsVisibleChanged .InvokeAsync(isVisible);
24.     }
25. }
26. [Parameter]
27. public EventCallback<bool> IsVisibleChanged { get; set; }
28.
```

The use will be as follows:

```
1. <MyComponent @bind-IsDarkMode="IsDarkMode"
2.     @bind-IsVisible="IsVisible" />
```

As we can see, each parameter will report the property name after the **@bind** directive so that the Blazor runtime can distinguish which property is referenced. This is a very important feature when

we have components that need different configuration.

Cascading parameters

When a parameter must be passed to all or multiple sub-contexts, the **[Parameter]** is not enough to ensure the robustness of data and the components too. This is due to the multiple jumps that the **[Parameter]** must do from all the components. Imagine having four or five levels of components and only the last nested one needs the data. With the **[Parameter]** technique we must have the involved property inside all the components in the chain even if they do not need that information. The solution is the **[CascadingParameter]** attribute that allows parameters to be passed down through multiple levels of components directly, without passing them into each level.

Try to figure out the following scenario:

We have a page **index.razor** that manage a numeric counter and it contains a component named **Container.razor** which contains a component called **Content.razor** which have to receive the number managed from **index.razor** page. Speaking with code, the index page should be as follows:

```
1. @page "/"
2. <CascadingValue Value = "Count">
3.     <Container />
4. </CascadingValue>
5.
6. @code {
7.     public int Count { get; set; } = 5;
8.     protected override void OnInitialized()
9.     {
10.         base.OnInitialized();
11.     }
12. }
```

The component **Container** will be as follows:

```
1. <h3>Container</h3>
2. <Content />
```

Finally, the **Content** component will be as follows:

1. `<h3>Content</h3>`
2. `<p>The value is: @Count</p>`
3. `@code {`
4. `[CascadingParameter] public int Count { get; set; }`
5. `}`

As we can see, the `<Content/>` tag in index page is surrounded by `<CascadingValue>` component with a parameter `Value` that is set to the property we want to pass to component. In this case, the `Container` component does not need to manipulate the data and there is no reason he has some references of this data in the code. The value is passed through the component to reach the `[CascadingParameter]` property in the `Content` component as well. The result will be as follows:

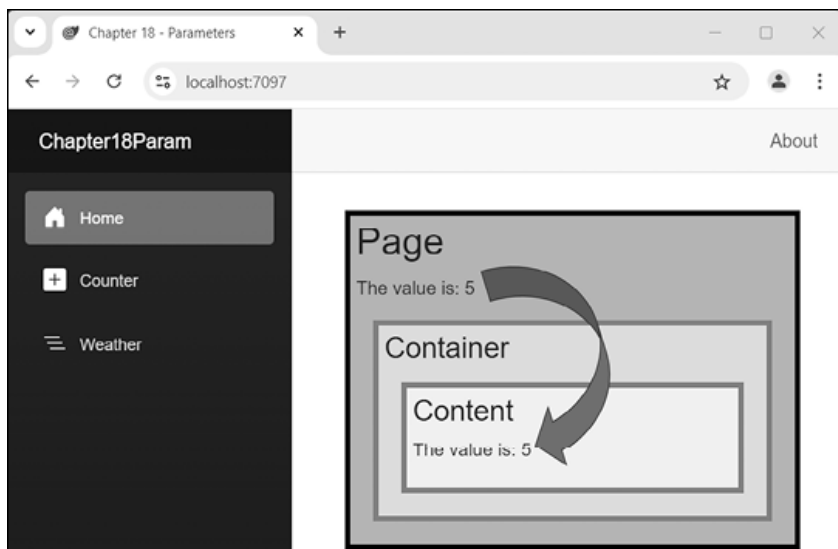


Figure 18.3: Cascading value result. The arrow represents the path of data.

EventCallback<T>

In the above nested components, we could have the necessity to pass back the result of an operation or user input from a child component to parent. It is possible with the `EventCallback<T>` in Blazor.

The `EventCallback<T>` is a generic struct and we can use as a type

passing as argument the datatype we want to transfer, as follows:

1. [Parameter]
2. `public EventCallback<bool> OnClickCallback { get; set; }`

In the following image, we can see how the **EventCallback** works in a multi component system.

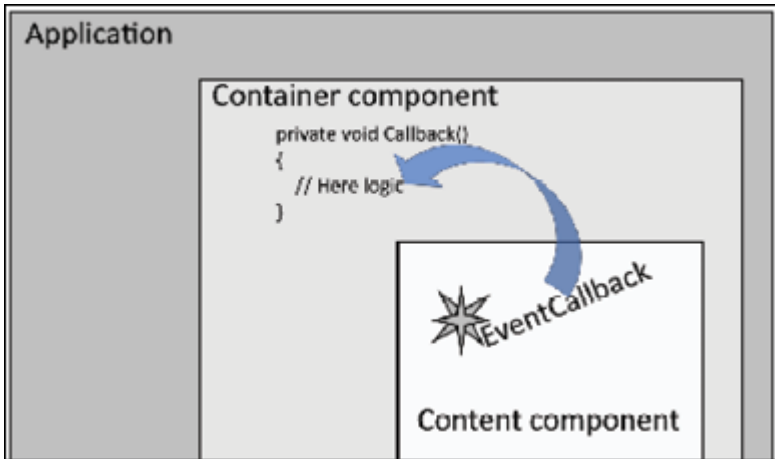


Figure 18.4: How `EventCallback<T>` works

In this scenario, we can write a small example of a Blazor Web App with nested components that interacts with each other through **CascadingParameters** and **EventCallback<T>**. For the first step, we can rewrite the component **Content** so that he can produce an event callback with a simple Boolean as follows:

1. `@rendermode InteractiveServer`
2. `<h3> Content </h3>`
3. `<p> The value is: @Count </p>`
4. `<p> Content value: @myCheck </p>`
5. `<p>`
6. `<button @onclick = "OnClickEvent">`
7. `Trigger a Parent component method`
8. `</button>`
9. `</p>`
10. `@code {`
11. `[CascadingParameter] public int Count { get; set; }`
12. `[Parameter]`

```

13. public EventCallback<bool> OnClickCallback { get; set; }
14. bool myCheck;
15. private void OnClickEvent()
16. {
17.     myCheck = !myCheck;
18.     OnClickCallback.InvokeAsync(myCheck);
19. }
20. }

```

Whenever we click on the button; we will produce an **EventCallback<T>** and in this specific case the **OnClickCallback** with a **boolean** parameter.

At this point, we can proceed with the **Container** component. This component will wrap and manage the **EventCallback<T>** callback properly. The code will be as follows:

```

1. @rendermode InteractiveServer
2. <h3>Container</h3>
3. <Content OnClickCallback = "ContainerCallBack" />
4. <p>Container value: @myCheck</p>
5. @code{
6.     bool myCheck;
7.
8.     private void ContainerCallBack(bool e)
9.         => myCheck = e;
10. }

```

As we can see, the component will pass to the **EventCallback<T>** parameter the function that will manage the return from the child component.

At last, the home page will be crafted as follows:

```

1. @rendermode InteractiveServer
2. <CascadingValue Value = "Count">
3.     <Container />
4. </CascadingValue>
5.
6. @code {
7.     public int Count { get; set; }
8.     protected override void OnInitialized()
9.     {

```

```

10.     Count = 5;
11.     base.OnInitialized();
12. }
13. }

```

The final application will have the following aspect:

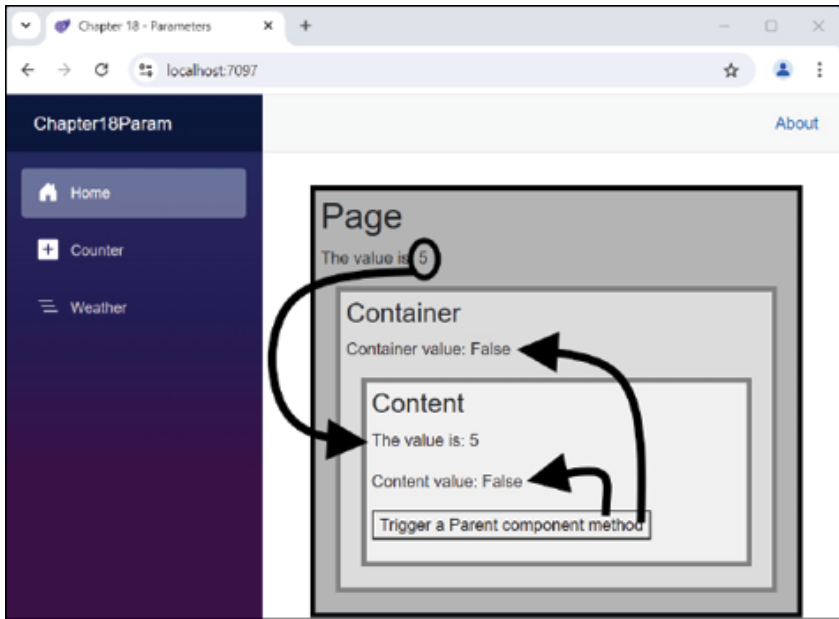


Figure 18.5: Application running with data path (arrows)

For each pressure on the button, the **Boolean** value shown, whose value is reproduced on both components, will be updated with the new value. Furthermore, at startup the **int** value is passed through **CascadingValue** directly to the **Content** component.

JavaScript interop

Blazor cannot directly access the **DOM** because it operates primarily on the server-side or in a WebAssembly-based sandbox. Blazor's code runs on .NET, not JavaScript, which means it doesn't natively interact with the **DOM**, which is a feature of the browser's JavaScript engine.

JavaScript interop is a feature that allows Blazor components to interact with JavaScript. While Blazor runs on .NET (either server-

side or via WebAssembly), it may still need to interact with the browser's **DOM** or other JavaScript libraries that are not available directly within .NET. JavaScript interop bridges this gap, enabling Blazor components to call JavaScript functions and vice versa.

Communication between Blazor and the **DOM** is managed through a **virtual-DOM** and **JavaScript interop**, which involves sending and receiving updates via JavaScript. This design ensures a separation of concerns, enhancing security, performance, and the ability to run .NET code in the browser, but it also limits direct, immediate **DOM** access that is typical in traditional JavaScript-based frameworks.

Calling JavaScript from Blazor

Blazor components can invoke JavaScript functions using the **IJSRuntime** service. This service provides methods like **InvokeVoidAsync**, for functions that do not return a value, and **InvokeAsync<T>**, for functions that return a value of type **T**.

Following is an example:

```
1. @inject IJSRuntime jsRuntime
2. <button @onclick="CallJavaScriptFunction">Click me</button>
3. @code {
4.     private async Task CallJavaScriptFunction()
5.     {
6.         await jsRuntime.InvokeVoidAsync
7.             ("console.log", "Hello from Chapter 18!");
8.     }
9. }
```

In this example, clicking the button triggers a C# method that calls a JavaScript **console.log** function.

Calling Blazor function from JavaScript

Through JavaScript interop we can use JavaScript code to invoke .NET methods on Blazor. On JavaScript side we use the **DotNet.invokeMethodAsync** function. This requires the method to be static (we could use a non-static method but a **DotNetObjectReference** instance is required) and annotated with **[JSInvokable]** on C# side as follows:

```

1. public class MyBlazorComponent
2. {
3.     [JSInvokable]
4.     public static string GetMessage()
5.     {
6.         return "Hello from Blazor!";
7.     }
8. }

```

The JavaScript function will be as follows:

```

1. <script>
2.     async function callDotNetMethod() {
3.         const message = await DotNet.invokeMethodAsync
4.             ('AppAssemblyName', 'GetMessage');
5.         console.log(message);
6.     }
7. </script>

```

The **AppAssemblyName** is necessary to well identify our Blazor application and must be exactly the application namespace we have defined.

The JavaScript function **callDotNetMethod** calls the Blazor method **GetMessage**, which returns a string.

Passing complex data

Both C# and JavaScript can pass complex data types (like objects) between each other because JSON serialization is used under the hood, the operations should be slow due to serialization and deserialization, but it makes easy to work with complex data.

Following is an example.

```

1. Person person = new() { Name = "John", Age = 30 };
2. await jsRuntime.InvokeVoidAsync("showPerson", person);

```

Then, a simple JavaScript function to manage this will be as follows:

```

1. function showPerson(person) {
2.     console.log(`Name: ${person.Name}, Age: ${person.Age}`);
3. }

```


Under the hood, when an object is passed from C# to JavaScript, Blazor serializes the C# object into a format that JavaScript can understand. This serialization is done in JSON, as it is a format that both .NET and JavaScript can easily interpret. The properties of the C# object are mapped to their equivalents in JSON, preserving the data types as closely as possible.

Asynchronous operations

Since Blazor operates asynchronously, JavaScript interop also supports asynchronous operations. The **InvokeAsync** method is inherently asynchronous, and JavaScript promises are seamlessly integrated. This functionality allows us to enhance Blazor's capabilities, providing for the following advantages:

- **DOM manipulation:** While Blazor abstracts away direct DOM access, certain complex manipulations require JavaScript mandatory.
- **Using JavaScript libraries:** Blazor can leverage existing JavaScript libraries like *jQuery*, *D3.js*, or *Chart.js*.
- **Browser APIs:** Accessing browser-specific APIs (e.g., **localStorage**, **geolocation**) requires the use of JavaScript.

There is no way to access these functionalities without using JavaScript and without using asynchronous methods. JavaScript Interop helps us to work with them.

Advanced use of JavaScript interop

In a normal Blazor application, when we use always C#, if we have to invoke a simple JavaScript function, the previous ways are enough. But if we have to interact with JavaScript libraries or wrap some browser functions, is not possible to use that approach. Blazor allows us to use a loading mode for a file through an interface named **IJSObjectReference**. To use that mode, we have to structure a JavaScript file with our functions and a C# class to wrap it. Let us see a real example. Imagine we want to display a modal on the screen using the Bootstrap library.

For the first activity, we have to build the JavaScript file. In this file, we have to define two functions, that is, the first for show the modal and the second to hide the modal. The two functions in

JavaScript will be as follows:

```
1. var myModal = {};  
2. export function showModal(id) {  
3.   myModal[id] = new bootstrap.Modal('#' + id);  
4.   myModal[id].show();  
5. }  
6. export function hideModal(id) {  
7.   myModal[id].hide();  
8. }
```

Now, we setup the Blazor project as **Blazor Web App** template and in the server project we add a **Services** folder which inside we add a class named **ModalService**, and the correspondent **IModalService** interface, able to manage the modal through JavaScript interop. It will be the following:

```
1. public class ModalService : IAsyncDisposable, IModalService  
2. {  
3.   private readonly IJSRuntime jsRuntime;  
4.   private IJSObjectReference? module = null;  
5.   public ModalService(IJSRuntime jsRuntime)  
6.     => this.jsRuntime = jsRuntime;  
7.   public async Task Init()  
8.     => module = await  
   jsRuntime.InvokeAsync<IJSObjectReference>  
9.     ("import", "./confirm.js");  
10.   public async Task ShowModal(string id)  
11.     => if (module is not null)  
12.       await module.InvokeVoidAsync("showModal", id);  
13.   public async Task HideModal(string id)  
14.     => if (module is not null)  
15.       await module.InvokeVoidAsync("hideModal", id);  
16.   public async ValueTask DisposeAsync()  
17.     => if (module is not null)  
18.       await module.DisposeAsync();  
19. }
```

The **Init()** function is responsible to import the JavaScript module into our application allowing us use their functions. If we call the **Init()** method multiple times, each call will re-import the JavaScript

module and assign a new instance of the module reference to the **module** variable. This can lead to issues because each time **Init()** is called, the previous reference is overwritten, potentially causing us to lose access to the earlier instance. Additionally, importing the module repeatedly could lead to unnecessary overhead, especially if the file is large or if the import operation has side effects. Furthermore, in an asynchronous context, calling **Init()** multiple times in quick succession might cause synchronization issues, making it difficult to determine which version of the module reference will be used in subsequent operations. To avoid these problems, it is a good practice to check if the module has already been initialized before attempting to import it again as follows:

```
1. public async Task Init()
2. {
3.     if (module is null)
4.         module = await
jsRuntime.InvokeAsync<IJSObjectReference>
5.         ("import", "./confirm.js");
6. }
7.
```

This ensures that the module is only imported once, preserving the original reference and preventing unnecessary operations.

At this point, we need a component to show as modal that will be named **Modal.razor** and it will be as follows:

```
1. @rendermode InteractiveServer
2. @inject IModalService ModalService;
3. <div id="id-modal"
4.     class="modal fade" tabindex="-1"
5.     aria-labelledby="modalLabel" aria-hidden="true">
6.     <div class="modal-dialog">
7.         <div class="modal-content">
8.             <div class="modal-header">
9.                 <h5 class="modal-title"
10.                    id="modalLabel"> @Title </h5>
11.                 <button type="button"
12.                    class="btn-close"
13.                    data-bs-dismiss="modal">
```

```

14.         aria-label = "Close" > </button>
15.     </div>
16.     <div class = "modal-body">
17.         @Message
18.     </div>
19.     <div class = "modal-footer">
20.         <button
21.             type = "button"
22.             class = "btn btn-secondary"
23.             @onclick = "OnCancel"
24.             data-bs-dismiss = "modal"> Cancel </button>
25.         <button
26.             type = "button"
27.             class = "btn btn-primary"
28.             @onclick = "OnConfirm"> Ok </button>
29.     </div>
30. </div>
31. </div>
32. </div>
33.
34. @code {
35.     [Parameter]
36.     public string Title { get; set; } = "Chapter 18";
37.     [Parameter]
38.     public string Message { get; set; } = "???";
39.     [Parameter]
40.     public EventCallback OnCancel { get; set; }
41.     [Parameter]
42.     public EventCallback OnConfirm { get; set; }
43.     protected override async Task OnAfterRenderAsync(bool
firstRender)
44.     {
45.         if (firstRender)
46.             if (ModalService is not null)
47.                 await ModalService.Init();
48.     }
49. }

```

This module will be used as modal in Bootstrap passing the main

selector of the div, the **id-modal**.

Finally, we have to call them and finalize this round. In the **Home.razor** we put the reference to our **Modal.razor** component and a button to open it.

The following razor code will be the **Home.razor** page:

```
1. @page "/"
2. @rendermode InteractiveServer
3. @inject IModalService ConfirmService;
4. <PageTitle> Home </PageTitle>
5. <button @onclick = "RequestDelete" > Open </button>
6. <Confirm OnConfirm = "ConfirmDelete"
7.     Message = "Welcome to JavaScript interop!" />
8. @code{
9.     protected override async Task OnAfterRenderAsync
10.         (bool firstRender)
11.     {
12.         if (firstRender)
13.             if (ConfirmService is not null)
14.                 await ConfirmService.Init();
15.     }
16.     public async Task RequestDelete()
17.     {
18.         if (ConfirmService is not null)
19.             await ConfirmService.ShowModal("id-modal");
20.     }
21.     public async Task ConfirmDelete()
22.     {
23.         if (ConfirmService is not null)
24.             await ConfirmService.HideModal("id-modal");
25.     }
26. }
```

Now, the project is ready to be launched and after the startup we can press the button to show the modal. The aspect will be as follows:

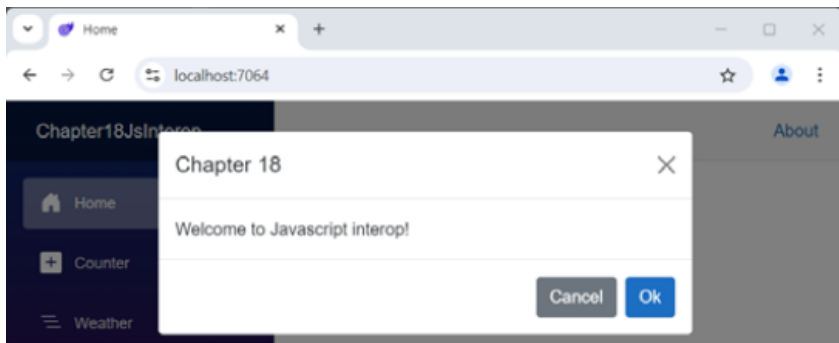


Figure 18.6: JavaScript modal application running

As we can see, we have two buttons on the modal connected with `EventCallback<T>` to functions on `Home.razor` page.

Basic state management

As discussed earlier in the book, the optimal approach for using an ASP.NET Core application is by implementing a stateless backend. However, throughout the application's lifecycle, there arises a necessity to store certain data temporarily. Unfortunately, there is no built-in mechanism to manage the application state natively. In a Blazor application, and in an ASP.NET Core application in general, when we leave a page, this state is lost. There are many techniques to avoid this problem. The best ways to manage the application state are the following:

- Store data in-memory
- Store data server side
- Store data in browser storage

Let us see an overview of them.

Store data in memory

The easiest way to manage the temporary state, is to store it in memory. By using a service class, we can manage state across different components by keeping data in memory. For instance, we could implement a state management service that tracks a user's preferences, like their choice of theme (dark or light mode for example). This service would store the data in memory, enabling it to be shared across various components within the application.

Create the state management service

We will create a class that will serve as the state management service. This class will store the data in memory and provide methods to access and modify the data as follows:

```
1. public class UserPreferencesService : IUserPreferencesService
2. {
3.     public string Theme { get; private set; } = "Light";
4.     public void SetTheme(string theme)
5.     {
6.         Theme = theme;
7.         NotifyStateChanged();
8.     }
9.     // Event to notify components when the state changes
10.    public event Action OnChange;
11.    private void NotifyStateChanged()
12.        => Dispatcher.CreateDefault().InvokeAsync(OnChange);
13. }
```

Obviously, we have to create the interface linked to the class as usual. This service class acts as a state manager, holding the current theme in memory and providing methods to change it. It also has an event (**OnChange**) that components can subscribe to in order to be notified when the state changes. Before to use the above service, we have to register the **UserPreferencesService** in the Blazor application's dependency injection container as usual. It is good to register it as a Singleton.

Use the service in a Blazor components

Now, let us create a Blazor component that interacts with the **UserPreferencesService** to allow the user to select a theme. The Blazor component allows the user to select a theme from a dropdown. When the theme is changed, the component updates the state via the **UserPreferencesService**, and the UI reflects the current theme. The component also subscribes to the **OnChange** event to automatically refresh the UI when the state changes as follows:

```
1. @page "/preferences"
2. @inject UserPreferencesService UserPreferences
3. <h3> User Preferences </h3>
```

```

4. <div>
5.   <label> Select Theme: </label>
6.   <select @onchange = "OnThemeChanged">
7.     <option value = "Light"
8.       selected = "@IsSelected("Light")"> Light </option>
9.     <option value = "Dark"
10.      selected = "@IsSelected("Dark")"> Dark </option>
11.   </select>
12. </div>
13. <p> Current Theme: @UserPreferences.Theme </p>
14. @code {
15.   private void OnThemeChanged(ChangeEventArgs e)
16.   {
17.     var selectedTheme = e.Value.ToString();
18.     UserPreferences.SetTheme(selectedTheme);
19.   }
20.   private bool IsSelected(string theme)
21.     => UserPreferences.Theme == theme;
22.   protected override void OnInitialized()
23.     => UserPreferences.OnChange += StateHasChanged;
24.   public void Dispose()
25.     => UserPreferences.OnChange -= StateHasChanged;
26. }

```

This example demonstrates how we can implement and use a simple state management solution in Blazor using an in-memory service. The **UserPreferencesService** can be expanded to manage more complex states as needed for our application.

Store data server side

In a Blazor application, storing data server-side is a good strategy for managing the application state, especially in scenarios where security, consistency, and scalability are fundamental aspects. By keeping data on the server, we can ensure that sensitive information is not exposed to the client, reducing the risk of unauthorized access or manipulation. This approach also allows for more consistent state management across different client sessions, as the server maintains the single source of truth. Server-side data storage is particularly beneficial in Blazor Server apps, where the application logic is

executed on the server, and only UI updates are sent to the client. Effective state management ensures that the application remains responsive and provides a seamless user experience, even as users navigate through different components or refresh the page.

Store data in browser storage

The browser storage is of two types, as follows:

- Local storage
- Session storage

The differences between these two types of browser storage lie in their lifespan, scope, and intended use. The only problem is that we cannot manage it directly from Blazor but we have necessary pass-through JavaScript interop as we have seen in precedent section **Advanced JavaScript interop**.

Local and session storage

Local storage is a client-side storage mechanism that retains data even after the browser is closed and reopened. This makes it particularly useful for storing information that needs to persist across multiple browsing sessions, such as user preferences, settings, or items in a shopping cart. Data stored in local storage are shared across all tabs, allowing consistent access to the stored information across different parts of the application. With a larger storage capacity, typically around 5-10 MB depending on the browser, local storage is well-suited for storing larger amounts of persistent data. The data remains available until it is explicitly deleted by the user or the application, making it a reliable option for long-term storage needs.

Session storage, in contrast, is designed for temporary storage of data that is only relevant for the duration of a page session. The data stored in session storage is automatically deleted once the browser tab is closed, ensuring that it does not persist beyond the current session. This makes session storage ideal for scenarios where data needs to be isolated to a single session or tab instance, such as preserving form inputs or user interactions while navigating within a specific tab. Unlike local storage, session storage is not shared across different tabs, which ensures that data is kept separate and

does not interfere with other sessions. While session storage typically offers the same storage capacity as local storage, its temporary nature makes it best suited for handling data that does not need to be retained beyond the active session.

JavaScript functions

To manage local storage and session storage, JavaScript provides specific functions that can be encapsulated in a JavaScript class, allowing us to use them with Blazor's interoperability features. The class will look as follows:

```
1. export function setItem(storageType, key, value) {
2.     const storage = storageType === 'session'
3.         ? sessionStorage : localStorage;
4.     storage.setItem(key, value);
5. }
6. export function getItem(storageType, key) {
7.     const storage = storageType === 'session'
8.         ? sessionStorage : localStorage;
9.     return storage.getItem(key);
10. }
11. export function removeItem(storageType, key) {
12.     const storage = storageType === 'session'
13.         ? sessionStorage : localStorage;
14.     storage.removeItem(key);
15. }
16. export function clear(storageType) {
17.     const storage = storageType === 'session'
18.         ? sessionStorage : localStorage;
19.     storage.clear();
20. }
```

We can load this file through a service class as seen in previous section *Advanced use of JavaScript interop* but we can use it **as is** importing the JavaScript file with the `<script>` tag.

The import with `<script>` tag of this JavaScript in Blazor is different according to the hosting model we are using.

Blazor Server

Edit file `Pages/_Host.cshtml` and add the following line after the

blazor.server.js line:

```
1. <script src = "javascript/browserStorage.js" > </script>
```

Blazor WebAssembly

Edit file **wwwroot/index.html** and add the following line after the **blazor.webassembly.js** line:

```
1. <script src = "javascript/browserStorage.js" > </script>
```

Blazor Web App

Edit file **App.razor** and add the following line after the **blazor.web.js** line:

```
1. <script src = "javascript/browserStorage.js" > </script>
```

This way, the Blazor project references the JavaScript code directly.

Service class

As usual we need a service class to manage the JavaScript and will be as follows:

```
1. public class BrowserStorageService : IBrowserStorageService
2. {
3.     private readonly IJSRuntime jsRuntime;
4.     private IJSObjectReference moduleTask = null;
5.
6.     public BrowserStorageService(IJSRuntime jsRuntime)
7.     => this.jsRuntime = jsRuntime;
8.
9.     public async Task Init()
10.    => moduleTask = await jsRuntime
11.        .InvokeAsync<IJSObjectReference>
12.        ("import", "./browserStorage.js");
13.
14.    public async Task SetItemAsync
15.        (string storageType, string key, object value)
16.    => await moduleTask.InvokeVoidAsync("setItem"
17.        , storageType, key
18.        , JsonSerializer.Serialize(value));
19.
20.    public async Task<T?> GetItemAsync<T>
21.        (string storageType, string key)
```

```

22. {
23.     var jsonValue = await moduleTask.InvokeAsync<string>
24.         ("getItem", storageType, key);
25.     return jsonValue == null
26.         ? default
27.         : JsonSerializer.Deserialize<T>(jsonValue);
28. }
29.
30. public async Task RemoveItemAsync
31.     (string storageType, string key)
32.     => await moduleTask.InvokeVoidAsync
33.         ("removeItem", storageType, key);
34.
35. public async Task ClearAsync(string storageType)
36.     => await moduleTask.InvokeVoidAsync("clear",
    storageType);
37. }

```

Use of browser storage in Blazor

To have an example of the use, we have to implement a simple component in Blazor able to manage all the JavaScript functions. It should be something like the following code:

```

1. @page "/storage"
2. @rendermode InteractiveServer
3. @inject IBrowserStorageService StorageService
4. <h3>Storage Example </h3>
5. <button @onclick="SetLocalStorage">Set Local Storage</
    button>
6. <button @onclick="GetLocalStorage">Get Local Storage</
    button>
7. <button @onclick="ClearLocalStorage">Clear Local Storage</
    button>
8. <p>@message</p>
9. @code {
10.     private string message;
11.
12.     protected override Task OnInitializedAsync()
13.     {

```

```

14.     if (StorageService is not null)
15.         StorageService.Init();
16.     return base.OnInitializedAsync();
17. }
18.
19. private async Task SetLocalStorage()
20. {
21.     await StorageService.SetItemAsync
22.         ("local", "myKey", "Hello, Blazor!");
23.     message = "Data set in Local Storage.";
24. }
25.
26. private async Task GetLocalStorage()
27. {
28.     var result = await StorageService
29.         .GetItemAsync<string>("local", "myKey");
30.     message = $"Retrieved from Local Storage: {result}";
31. }
32.
33. private async Task ClearLocalStorage()
34. {
35.     await StorageService.ClearAsync("local");
36.     message = "Local Storage cleared.";
37. }
38. }
39.

```

With this code we can manage the application state directly in the browser storage. We can write and read the local storage or session storage from the whole application without any problems, but we have to take note that the data will be in the client's browser, and we do not have to store sensible data.

The code can be accessed from the [github link](#).

Conclusion

In this chapter, we have seen advanced Blazor techniques, including form validation, parameter passing for dynamic navigation, JavaScript interop for extended functionality, and basic state

management. These tools enhance user data accuracy, application robustness, and enable building sophisticated, high-performing Blazor applications.

In the next chapter, we will address how to properly structure a project for testing and how to efficiently debug it to identify bottlenecks and optimize the overall performance of our system.

Points to remember

- **State management:** Manages and preserves data consistency throughout the app.
- **JavaScript interop:** Integrates JavaScript with other frameworks for advanced features.
- **Browser storage:** A client-side storage in web browsers for saving data locally, such as `localStorage` or `sessionStorage`.

Multiple choice questions

1. What is the primary purpose of input form validation?
 - a. To improve the visual design of forms
 - b. To ensure user data is accurate and meets required formats
 - c. To automatically submit the form
 - d. To store user data in the database
2. What does state management primarily help with in an application?
 - a. Enhancing the speed of the application
 - b. Designing the user interface
 - c. Handling and storing application data efficiently
 - d. Debugging the application
3. Which of the following is a key benefit of JavaScript interop?
 - a. It simplifies server-side logic
 - b. It improves the security of the application
 - c. It automatically manages the state of the application
 - d. It allows for advanced functionality by integrating

JavaScript with the application

4. **Why is passing parameters through application pages necessary?**
 - a. Because we can maintain user data and context across different pages
 - b. Because we can speed up the page loading time
 - c. Because we can automatically validate user inputs
 - d. Because we can manage the application's overall performance
5. **JavaScript interop is particularly useful when we need to:**
 - a. Improve the application's backend performance
 - b. Integrate existing JavaScript exclusive functions with the application
 - c. Automate input form validation
 - d. Manage database transactions

Answer key

Key terms

- **Validation:** Ensuring user input meets criteria before processing within the application.
- **Parameters:** Transferring data across application pages for dynamic content management.
- **State:** Managing and persisting data to maintain consistent application behavior.
- **Forms:** Structured user inputs collected for processing and application functionality.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 19

Debugging, Testing, and Performance Tuning

Introduction

In this chapter, we will explore the essential skills of debugging and performance tuning, crucial for producing robust and efficient code. We begin by introducing debugging, focusing on how to systematically identify, diagnose, and resolve errors in our software. Next, we will cover various testing strategies to catch issues early and ensure our code behaves as expected.

We will also discuss debugging best practices to streamline our approach, making the process more effective and less time-consuming. Finally, we will delve into performance tuning, learning how to optimize our code for faster, more efficient execution.

Structure

In this chapter, we will cover the following topics:

- Introduction to debugging
- Testing strategies
- Debugging best practices
- Performance tuning

Objectives

By the end of this chapter, readers will have a comprehensive understanding of debugging, equipping us with the skills to systematically identify and resolve errors in our software. We will also gain insight into the structure of software testing, learning how to implement strategies that ensure our code functions as intended. Additionally, the chapter will guide us in enhancing our applications by applying the best solutions for memory optimization and algorithm optimization, ultimately improving both performance and efficiency. By the end of this chapter, we will be equipped with the knowledge to enhance both the reliability and performance of our software, elevating our development skills.

Introduction to debugging

Debugging is a fundamental aspect of the software development lifecycle, particularly when working with complex frameworks like ASP.NET Core. In this section we will introduce the debugging process in ASP.NET Core, focusing on essential tools, techniques, and best practices that help developers efficiently identify and resolve issues.

Debugging in ASP.NET Core

At its core, debugging is the process of identifying and fixing bugs or errors, flaws, or unwanted behaviors in code. In ASP.NET Core, debugging typically involves working within an **integrated development environment (IDE)** to track down issues in web applications, whether they arise during development or after deployment. ASP.NET Core, being a high-performance and cross-platform framework, offers robust support for debugging. The framework's modular architecture, middleware pipeline, and dependency injection system provide powerful mechanisms for

building web applications, but they can also introduce challenges when issues arise. Effective debugging ensures that applications run smoothly, perform well, and provide a reliable user experience.

Debugging tools

Several tools are available to assist with debugging in ASP.NET Core. These tools are integrated with popular IDEs, such as Visual Studio and Visual Studio Code, to provide a seamless debugging experience. Visual Studio offers a rich debugging environment, including features like breakpoints, watches, and an immediate window for evaluating expressions. It also supports advanced debugging capabilities such as step-through debugging, variable inspection, and conditional breakpoints.

The integrated console and logging features in ASP.NET Core also play a crucial role in debugging. The framework includes a powerful logging system that helps trace and diagnose issues by capturing logs at various levels, for example Information, Warning, Error). These logs can be viewed directly in the console or stored for later analysis. Additionally, ASP.NET Core supports remote debugging, which allows us to debug applications running on remote servers or in cloud environments. This feature is particularly useful for diagnosing issues that only occur in specific production environments.

Debugging techniques

Debugging in ASP.NET Core involves several techniques that help systematically identify and resolve issues. One of the most fundamental techniques is using breakpoints, which are markers placed in the code where execution pauses, allowing developers to inspect the current state of the application.

Debugging using breakpoints is the primary debugging technique used by every developer. In Visual Studio 2022, this technique offers many advanced features, often underestimated, that allow for efficient debugging. Beyond simply stopping program execution, it is possible to:

- **Modify the execution pointer** to skip or re-execute code.
- **Set conditions** to break execution only in specific situations.

- **Use temporary breakpoints**, useful for stopping the code only once.
- **Log actions** or track hit counts without pausing execution.
- **Monitor variables in memory** using data breakpoints.
- **Break on specific exceptions**.

With tools like the **Breakpoints** window, Visual Studio 2022 makes it easy to configure and manage these advanced features. Breakpoints are essential for isolating and understanding problematic code sections.

Another essential tool for debugging the RESTful APIs is the **.http** or **.rest** files used directly within Visual Studio 2022. These files allow us to write and send HTTP requests to test API endpoints easily and efficiently. With these files, we can define HTTP requests like GET, POST, PUT, and more, and send them directly from the editor. The response is displayed immediately, allowing us to quickly verify the behavior of endpoints without leaving the development environment. This functionality makes the process of debugging APIs more streamlined and integrated, enabling us to focus on development and testing without disrupting our workflow.

Additionally, we can easily navigate to the related source code, facilitating troubleshooting and code adjustments in response to test results. We create a file with a **.http** extension, where we can write HTTP requests as follows:

```
1. GET https://bpb.books.com/posts/1
```

Or else, a more verbose request like the following POST request:

```
1. POST https://bpb.books.com/posts
2. Content-Type: application/json
3. {
4.   "title": "foo",
5.   "body": "bar",
6.   "userId": 1
7. }
```

Directly on the written request above, there is a little link **Send request** to execute it. Visual Studio will display the response directly beside the code as we can see in the following figure:

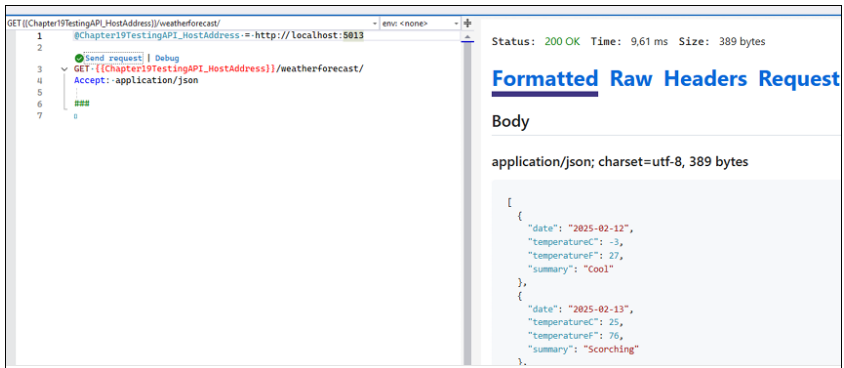


Figure 19.1: Http request file in action

We can also use variables, following is an example:

1. ###
2. @baseUrl = https://localhost:7167
3. GET {{baseUrl}}/WeatherForecast

When the request has passed, the little green circle will appear, otherwise a red circle with a cross will appear. This feature streamlines and speeds up API testing directly within our development environment.

Endpoints explorer is a powerful tool present in Visual Studio 2022 from version 17.6 able to find all the endpoints in the solution and explore them. As we can see in the [Figure 19.2](#), we can easily activate the tool, which provides an overview of all the endpoints. We can test the endpoints directly from the tool by right-clicking on the endpoint and selecting the **Generate request** option. This action creates a new HTTP request file that allows us to thoroughly test the endpoint. The tool also allows us to navigate to the corresponding code. By right-clicking on the endpoint and selecting **Open in the editor**, we are directed to the corresponding code file in the Visual Studio solution.

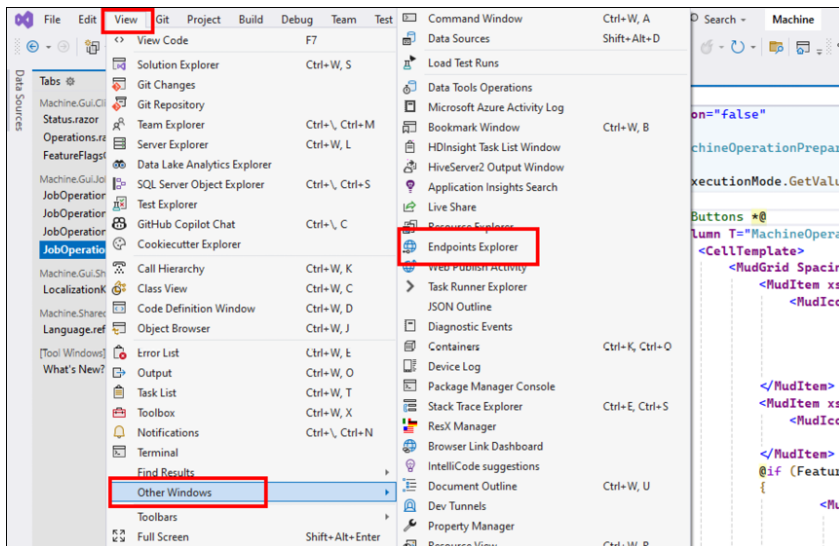


Figure 19.2: Endpoints explorer menu

This process allows us to diagnose and optimize our ASP.NET Core application using the powerful analysis tools provided by Visual Studio 2022.

When the application flow is particularly complex as real-time applications or IoT cockpits, the breakpoints could not be a solution. In this scenario, we can setup a logging strategy able to wrap the unexpected condition. In the logging trace, there will be the timestamp, a message that can help us trace the specific function, the expected value (if any) and the effective value. This technique will help for a lot of scenarios where we cannot use breakpoints but we have to delete them before production. In any case, every instrument that we put to monitor our application, we perturb the system. Be careful using that kind of instruments to avoid behavior discrepancy, in other words, let us avoid making the cure worse than the disease.

Testing strategies

Testing is an essential part of software development process, ensuring that the application behaves as expected and meets both functional and non-functional requirements. In ASP.NET Core, testing strategies are crucial for maintaining the quality and

reliability of all applications as they evolve. This section explores various testing strategies that can be employed in ASP.NET Core, providing a foundation for building robust and resilient web applications.

Importance of testing in ASP.NET Core

In *Chapter 12, Automated Testing* we have addressed the topic of automatic testing in all of the aspects. We have encountered unit testing, integration testing, end-to-end testing etc. and we have concluded that testing in ASP.NET Core is essential for identify bugs and issues early in the development cycle, reducing the cost and effort required to fix them later. Moreover, well-tested applications are more maintainable, as tests provide a safety net when making changes or refactoring code. Testing also ensures that the application meets its requirements and performs reliably under various conditions. Let us resume all the test methodologies so that we can choose the best strategy for our project.

Unit testing

Unit testing is the most basic form of testing, focusing on individual components or functions in isolation to ensure they work correctly. In ASP.NET Core, unit tests are commonly written using frameworks like *xUnit*, *MSTest*, *NUnit*, or *bUnit* for Blazor.

ASP.NET Core's modular architecture simplifies unit testing, allowing us to test controllers, services, and other components independently. By using mocking frameworks like **Moq**, we can isolate the units being tested by simulating their dependencies, ensuring the test focus solely on the behavior of the unit test. Well-crafted unit tests not only validate component logic but also document expected code behavior.

Integration testing

While unit testing focuses on individual components, integration testing verifies that different parts of the application work together as expected. In ASP.NET Core, integration tests involve checking the interactions between multiple components, such as services, databases, and APIs, ensuring they function correctly when combined.

ASP.NET Core offers tools like the **WebApplicationFactory** class, which creates an in-memory test server to run tests without deploying the application. This allows us to test the entire application pipeline, including middleware, controllers, and data access layers, in a controlled environment. Integration tests help identify issues arising from component interactions, ensuring the application functions as a cohesive unit.

End-to-end testing

End-to-end testing, or functional testing, verify an application's functionality from the user's perspective by simulating real-world scenarios. In ASP.NET Core, these tests cover UI interactions, API calls, and overall workflows. End-to-End test suite execution often requires that the entire system is up and running, which can lead to high resource demands.

While [Chapter 12, Automated Testing](#) introduces basic **xUnit** tests, tools like **Selenium** or **Playwright** offer advanced browser-based testing by simulating user interactions. ASP.NET Core's test server can also be used for functional testing of APIs and backend components. End-to-end tests ensure the application meets requirements, providing a seamless user experience and preventing regressions by covering critical user journeys.

Performance testing

Performance testing assesses how an ASP.NET Core application performs under various conditions, like high user load or limited resources, aiming to identify bottlenecks and optimize resource usage.

ASP.NET Core offers tools like **BenchmarkDotNet** for micro-benchmarking to measure application performance. These tests evaluate response times, throughput, and resource utilization, providing insights into the application's scalability and efficiency. Incorporating performance testing into development ensures that the application remains responsive and reliable under heavy load. In the **performance tuning** section, we will refine these techniques further.

Continuous testing

Continuous testing integrates testing into the CI/CD pipeline, ensuring tests run automatically with each code commit or deployment. In ASP.NET Core, this can be achieved using tools like Azure DevOps, GitHub Actions, or Jenkins. Just a quick note: in medium to large teams, this approach can lead to significant energy waste due to unnecessary commits. It is crucial to collaborate with those managing the pipeline to establish a solid strategy that avoids waste and prevents unnecessary pipeline executions.

By automating unit, integration, functional, and performance tests, continuous testing ensures the application remains stable, quickly identifying and addressing issues introduced by new changes. This approach provides rapid feedback and reduces the risk of deploying buggy code to production. Continuous testing is an essential part for maintaining quality and reliability in agile development.

In general, testing strategies are essential for developing reliable ASP.NET Core applications. Combining unit, integration, functional, and performance testing ensures that applications meet requirements and perform well. Integrating these tests into a continuous testing strategy further enhances application stability, allowing for rapid iteration and confident deployment. As ASP.NET Core evolves, robust testing practices will continue to be vital for successful software development.

Debugging best practices

To ensure a smooth debugging process in ASP.NET Core, it is important to follow best practices that make identifying and resolving issues easier. As we said in first chapter of this book, code is not for machines but for humans. Writing readable code is the foundation of effective debugging. Clear, well-organized code is easier to debug, so it is essential to use meaningful variable names, consistent formatting, and break down complex functions into smaller, manageable units. Proper use of configuration is also crucial. ASP.NET Core's configuration system should be utilized to manage settings that can be changed without modifying code. This includes enabling or disabling debugging modes based on the environment (development, staging, production).

Testing locally before deploying is another best practice. Running

and debugging the application locally before deploying to a production environment allows developers to catch and fix issues early. In the development environment, enabling detailed error messages is beneficial for debugging. This can be done by configuring the **DeveloperExceptionPage** middleware to display the full stack trace and error details. Sometime to run application locally are necessary other services that we cannot deploy on our laptop. In this case, Docker will help us to have a proper ecosystem to run and test locally our application or part of it. Strategic logging is also vital; implementing logging that captures critical events and errors without overwhelming the system with unnecessary information is essential for effective debugging.

Regularly updating dependencies is another important practice. Keeping the ASP.NET Core framework and related dependencies up to date ensures that the latest bug fixes and improvements are applied, which can prevent or resolve issues.

Performance tuning

In the previous section, we saw the *Performance testing* and how it is necessary. In this section, we will see how really performance tuning is a critical aspect of developing high-quality ASP.NET Core applications. It involves optimizing various components of an application to ensure that it runs efficiently, responds quickly, and can handle increasing loads. Performance tuning is not just about making an application faster; it is also about ensuring that it scales well and uses resources effectively. This section explores the key principles and strategies for performance tuning in ASP.NET Core, with a particular emphasis on identifying performance bottlenecks, including techniques for discovering memory leaks.

Identifying performance bottlenecks

Identifying performance bottlenecks is the first and most crucial step in performance tuning for ASP.NET Core applications. Bottlenecks can manifest in various forms, such as memory leaks, inefficient algorithms, or uneven load distribution across resources. To address these issues effectively, we must utilize a combination of monitoring tools, profiling techniques, and optimization strategies.

Memory leak discovery techniques

Memory leaks occur when an application unintentionally retains memory that it no longer needs, leading to increased memory consumption over time. In long-running applications, such as web servers, memory leaks can cause the application to slow down, crash, or consume excessive system resources. Identifying and fixing memory leaks is essential for ensuring the performance and stability of an ASP.NET Core application. In ASP.NET Core, we can use several diagnostic tools to monitor memory usage and identify leaks. Tools like **dotnet-counters** and **dotnet-dump** provide real-time monitoring and allow the capture of memory dumps for in-depth analysis. The *Visual Studio Performance Profiler* includes a *Memory Profiler* that tracks memory allocation, identifies objects that are not being garbage collected, and helps pinpoint potential memory leaks. Additionally, analyzing **garbage collection (GC)** logs offers insights into memory allocation patterns and helps trace leaks to specific parts of the code. By enabling GC logging, we can observe how the garbage collector manages memory and identify objects that persist longer than expected.

Another useful technique is *heap snapshot analysis*, where we take snapshots of the managed heap at different times during the application's execution. By comparing these snapshots, we can identify objects that are unexpectedly growing in size or number, which often indicates a memory leak. These snapshots, combined with tools like *Visual Studio's Diagnostic Tools*, allow developers to track down and eliminate unnecessary references that cause memory retention.

Blazor Webassembly memory management

In *Chapter 17, Building Responsive User Interfaces with Blazor* we have seen what is Webassembly and how it is used as part of Blazor applications. Only to summarize, Blazor WebAssembly enables code execution directly in the user's browser. Therefore, it is of fundamental importance to properly manage memory to avoid blocking on the user's device, ensuring smooth performance and a responsive experience. Given the constraints of the WebAssembly platform, a well-defined memory management strategy is essential for maintaining the efficiency and reliability of these applications.

Unlike traditional .NET applications, Blazor WebAssembly operates within the WebAssembly platform using its specific sandbox, which uses a distinct memory model. Specifically, WebAssembly employs a contiguous block of memory known as *linear memory*, where all memory operations are conducted. While Blazor WebAssembly relies on the .NET runtime's garbage collector, the limited memory available in a web environment necessitates careful and efficient memory usage.

One important consideration is avoiding large object allocations, which can lead to memory fragmentation and increase the overhead of garbage collection. To mitigate this, it is beneficial to reuse objects wherever possible, rather than frequently creating new ones. In this case, ASP.NET Core assists us with dependency injection, allowing us to avoid creating duplicate objects by adding *Singleton* or *Scoped* objects that are available throughout the entire application, thereby improving memory efficiency.

Another critical aspect of memory management is handling the lifetimes of Blazor components properly. Blazor components follow a specific lifecycle, and managing their lifetimes correctly is essential for optimizing memory usage. Implementing the **IDisposable** interface on components and services allows for the proper release of resources when they are no longer needed. For example, if we inject a service class as usual and, in the component, we add a subscription to a service event, we must detach it in the dispose function. If we do not unsubscribe callbacks and repeatedly call the same component, we will face memory leaks and unpredictable behavior due to retained references and uncontrolled memory growth. Furthermore, it is important to avoid holding onto references to components longer than necessary, as this can prevent garbage collection and lead to increased memory consumption.

The choice of data structure also plays a significant role in memory management. Using efficient collections such as **List<T>** or **Array** can help reduce memory usage compared to more memory-intensive structures like **Dictionary<T>**. Additionally, opting for immutable data structures can help prevent unintended modifications, which in turn can contribute to more efficient memory use.

Data binding in Blazor, can also introduce memory overhead if not used carefully. Utilizing one-way binding where possible is a

strategy to reduce memory consumption. For scenarios where two-way binding is necessary, optimizing by limiting the frequency of updates and judiciously using events can help manage memory more effectively.

JavaScript interop is a powerful feature of Blazor WebAssembly, but it can lead to memory leaks if not managed properly. When creating JavaScript objects from Blazor, it is essential to ensure they are disposed of correctly to prevent leaks. Moreover, minimizing the number of interop calls can reduce the associated memory overhead.

Regular monitoring and profiling of memory usage are essential practices for identifying and resolving memory issues. Tools such as browser developer tools can be used to monitor memory usage and detect leaks in real-time. Additionally, .NET memory profiling tools can provide deeper insights into memory consumption patterns within a Blazor application.

Optimizing state management is another key factor in controlling memory usage, especially in complex applications. Using dedicated state containers to manage shared state can improve memory efficiency. Furthermore, favoring immutable state objects helps prevent unintended modifications, contributing to better overall memory management.

Lazy loading is an effective strategy for managing memory by loading resources only when they are needed. This approach can be applied both to components and data, reducing the initial memory footprint and improving the application's performance by loading only the necessary resources at the moment.

Algorithm optimization

Algorithmic inefficiencies are another significant source of performance bottlenecks. Poorly optimized algorithms can lead to slow response times, excessive CPU usage, and overall inefficiency. To address these issues, developers often start by profiling and benchmarking their code to identify slow methods and resource-intensive operations. Profiling tools such as *Visual Studio's Performance Profiler* or *BenchmarkDotNet* provide valuable data on how much time and resources are consumed by specific code

segments. With this data, we can focus their optimization efforts where they will have the most impact. Understanding the complexity of algorithms is crucial in this process.

An algorithm can have two ways of complexity, that is, *time complexity* and *spatial complexity*, as follows:

- **Time complexity:** Indicates the number of operations performed by the algorithm relative to the input size.
- **Space complexity:** Indicates the amount of memory required by the algorithm based on the input size.

To evaluate the complexity of an algorithm, is commonly used the Big-O notation. Big-O notation is a mathematical concept used to describe the upper bound of an algorithm's time or space complexity. It expresses how the runtime or memory usage scales relative to the input size, focusing on the most significant factor that dominates as the input grows, helping to evaluate an algorithm's efficiency. Low complexity algorithms have **O(n)** complexity and indicates that the complexity is linear. High-complexity algorithms, such as those with **O(n²)** time complexity, can cause significant performance degradation as data size increases, but algorithms with **O(n log n)** time complexity are perform significantly better. We often refactor these algorithms to reduce their complexity, for instance, by replacing a quadratic-time algorithm (for instance two **for** nested cycles) with a linear-time one (a single **for** or linear execution). Directly linked to algorithms there are data structures that should optimize too. For example, using a **Dictionary<T>** or a **HashSet<T>** instead of a **List<T>** can greatly improve lookup times, making the application more responsive. Dictionaries and hash tables offer (almost) constant access time, whereas lists have linear access time in the worst-case scenario. Reducing redundant operations, such as eliminating repeated calculations, further enhances algorithm efficiency. Caching expensive computations or query results can also reduce processing times, while memorization (storing precomputed results) can be beneficial in scenarios where the same calculations are performed repeatedly.

By addressing these performance bottlenecks through careful monitoring of memory usage, optimization of algorithms, and implementation of effective load balancing strategies, we can

significantly improve the performance and scalability of our ASP.NET Core applications.

Optimizing application code

Once performance bottlenecks, including memory leaks, have been identified, the next step is to optimize the application code. This involves reducing unnecessary computations, efficiently using asynchronous programming, optimizing loops and collections, and minimizing memory allocations. Specifically addressing memory leaks, developers should focus on proper resource management, ensuring that objects are disposed of correctly, and avoiding unnecessary object retention.

Daily tuning

Performance tuning is an ongoing process, much like tending to a garden where regular care and attention are essential. As we have seen in [Chapter 7, Architectural Principles](#), the software gardening is a fine tuning of each part of our application. Just as a gardener must continuously water, prune, and monitor their plants to ensure a thriving garden, developers must regularly test and monitor their software to maintain optimal performance. Performance tests should be conducted to simulate real-world usage, and monitoring tools should be used to keep track of memory usage, CPU load, and other critical metrics. Continuous monitoring, akin to the vigilant eye of a gardener, allows us to detect memory leaks and other performance issues early, preventing them from becoming weeds that could choke the user experience. Just as a well-maintained garden flourish, so too will software that is carefully monitored and tuned.

Conclusion

In this last chapter of the book, we developed essential skills to improve the quality and efficiency of our software. We explored debugging techniques to systematically identify and resolve errors, and we learned about software testing strategies to ensure our applications function correctly. Additionally, we focused on performance tuning through memory and algorithm optimization,

enhancing our applications' speed and efficiency. These practices will be crucial as we build high-quality, reliable software in the future.

Points to remember

- **Debugging essentials:** techniques to identify and resolve code errors efficiently.
- **Testing strategies:** tests early to ensure code functions as expected.
- **Big-O notation:** Big-O notation describes algorithm efficiency as input grows.
- **Memory leak:** memory leak occurs when unused memory is not released.
- **Software quality:** debugging, testing, and optimization for high-performing applications.

Multiple choice questions

1. **What is the primary purpose of debugging?**
 - a. To write new code
 - b. To identify and fix errors in the code
 - c. To create documentation
 - d. To test new features
2. **What is the best practice when encountering a difficult bug?**
 - a. Ignore it and it will resolve itself
 - b. Add more features to the code
 - c. Simplify the problem and break it down
 - d. Delete the problematic code
3. **What is the main goal of performance tuning?**
 - a. To increase the size of the application
 - b. To improve code readability
 - c. To make the application run faster and more efficiently

d. To add more features

4. Which of the following best describes unit testing?

- a. Testing individual components or functions
- b. Testing the entire application at once
- c. Testing the application's user interface
- d. Testing the application's performance under stress

5. What is a common outcome of memory optimization?

- a. Increased application size
- b. Reduced application performance
- c. More complex code structure
- d. More efficient use of system resources

Answer key

Key terms

- **Debugging:** Identifying and fixing errors in code.
- **Testing:** Validating code to ensure correct functionality.
- **Unit:** Smallest testable part of an application.
- **Algorithm:** Step-by-step procedure for solving problems.
- **Memory:** Storage used by applications during execution.
- **Efficiency:** Achieving desired outcomes with minimal resources.
- **Best-practices:** Proven techniques for effective software development.
- **Performance:** How fast and efficiently software runs.
- **Reproduction:** Replicating an issue for debugging purposes.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



Index

A

- acceptance testing [300](#)
- access control lists (ACLs) [315](#)
- Advanced Encryption Standard (AES) encryption [336](#)
- Agile [171](#)
- antipattern [162](#), [163](#)
 - big ball of mud [163](#)
 - God Object antipattern [164](#)
 - Golden Hammer [165](#)
 - lava flow [166](#)
- Application Insights [356](#)
- application programming interfaces (APIs) [87](#)
- architectural patterns [166](#), [167](#)
 - architectural principles [167](#)
- architectural principles
 - modularity [167](#), [168](#)
 - resilience [170](#)
 - scalability [168](#), [169](#)
- architecture decision records (ADRs) [173](#)
 - benefits [174](#)
 - post-deployment care [177](#)
 - software deployment [177](#)
 - software gardening [178](#)
 - software maintenance [177](#), [178](#)
 - software testing [176](#), [177](#)
 - structure [174](#)
 - system design [175](#), [176](#)
 - usage [174](#)
- Arrange-Act-Assert [295](#)
- ASP.NET Core
 - overview [2](#)
 - PC setup for [2-5](#)
- ASP.NET Core application
 - main parts [2](#)
- ASP.NET Core controller-based API [96](#), [97](#)
 - attributes [97](#), [98](#)
 - error status codes, returning [99](#), [100](#)
 - source parameter inference, binding [98](#), [99](#)

- ASP.NET Core minimal API [100](#), [101](#)
- ASP.NET Core project [5](#)
 - modifying [8](#), [9](#)
 - structure [34](#)
- ASP.NET Core Web API [94](#)
 - ASP.NET Core controller-based API [96](#)
 - controllers [94-96](#)
 - endpoints [94-96](#)
- ASP.NET Core Web App [6-9](#)
- attribute routing
 - with templates [123](#)
- authentication [330](#)
 - with JWT token [330](#)
- AuthN [315](#)
- authorization [330](#)
- authorization management [321](#), [322](#)
 - implementing [322](#)
- AuthZ [315](#)
- automated testing [286](#), [287](#)
 - test types [287](#)

B

- backend architecture layering [179-182](#)
 - benefits [183](#), [184](#)
 - Clean Architecture, as separated applications [182](#), [183](#)
 - essence of layering [180](#), [181](#)
- backend security
 - Application Insights [321](#)
 - best practices [314](#)
 - C-I-A [314](#)
 - configuration and secrets, securing [316](#)
 - configuration files, protecting [316](#), [317](#)
 - logging and monitoring [319](#)
 - middleware, using [320](#), [321](#)
 - monitoring [320](#)
 - sensitive data, encrypting [317-319](#)
 - Serilog logging, implementing [319](#), [320](#)
 - three Au's [315](#)
- black-box testing [293](#)
 - advantages [293](#)
 - implementation [294](#)
 - limitations [293](#)
- Blazor [376](#)
 - advantages [376-378](#)
 - syntax rules [378-380](#)
- Blazor Hybrid [390](#)
 - with MAUI [395](#), [396](#)

- with Windows Forms [394](#), [395](#)
 - with WPF [390-394](#)
- Blazor InteractiveAuto [399](#)
- Blazor InteractiveServer [398](#)
- Blazor InteractiveWebAssembly [398](#)
- Blazor JS scripts
 - location [37](#)
- Blazor render modes [397](#), [398](#)
- Blazor Server [233](#), [378-381](#)
 - features [381-384](#)
- Blazor StreamRendering attribute [399](#)
- Blazor United [400](#)
- Blazor Web App project [34](#)
 - client project structure [36](#)
 - middleware [38](#), [39](#)
 - server project structure [34-36](#)
- Blazor WebApp template [10](#)
- Blazor WebAssembly [378](#), [384](#)
 - architecture [387](#)
 - performance and optimization [389](#), [390](#)
- Bounded Context [73](#)

C

- CascadingParameters [210](#)
- catch-all parameters [119](#)
- catch-all route template [119](#)
- Clean Architecture [68](#), [69](#)
 - implementing [70-72](#)
- Client project [36](#)
 - pages folder [36](#)
 - wwwroot folder [37](#)
- client-side rendering (CSR) [34](#)
- client side usage [334](#)
- Command Pattern [44](#)
 - implementing [44-46](#)
- Command Query Responsibility Segregation (CQRS) [223-225](#)
 - application [228](#)
 - command handlers, modifying [231](#), [232](#)
 - domain models [228](#)
 - event sourcing, adding [230](#)
 - event store [230](#), [231](#)
 - implementing [226](#)
 - implementing, in practice [236](#)
 - infrastructures [228](#), [229](#)
 - integrating, with backend services [233](#), [234](#)
 - product management [226](#)
 - projection, in event sourcing [232](#), [233](#)

- project structure [226](#)
- results, visualizing [233](#)
- UI components, building [234-236](#)
- Web API [227](#)
- communication server-client, SignalR
 - long polling [261-263](#)
 - message queues [256](#), [257](#)
 - named pipes [255](#), [256](#)
 - server-sent events (SSE) [259-261](#)
 - shared memory [257-259](#)
 - WebSockets [263-265](#)
- complexity [11](#), [12](#)
 - causes [13](#)
 - multiple definitions [11](#)
 - recognizing [12](#), [13](#)
 - rules to avoid [13](#), [14](#)
 - technical debt [12](#)
- component-based architecture [46](#), [47](#)
 - TaskManagerApp component [47](#)
 - TaskModel class [48](#), [49](#)
- configuration and dependencies
 - handling, in containerized environment [370](#), [371](#)
- containerization [360](#)
 - achieving [362](#)
 - benefits [361](#)
- containerized website
 - creating [363-366](#)
- continuous integration (CI) pipelines [249](#)
- CQRS with event sourcing (CQRS-ES) [223](#)
- Create, Read, Update, Delete (CRUD) operations [101](#)
- cross-platform native applications [376](#)
- cross-site request forgery vulnerability [340](#), [341](#)
 - mitigating [341](#)
- cross-site scripting vulnerability [339](#), [340](#)
- cross-site scripting (XSS) attacks [324](#)
 - protecting against [324](#)
- custom middleware
 - creating [149](#), [150](#)
 - extension method, creating [151](#)
 - for short circuit in pipeline [152](#), [153](#)
 - middleware class, defining [150](#), [151](#)
 - registering, in pipeline [151](#), [152](#)

D

- data access layers (DAL) [141](#)
- data encryption
 - principles and rule [334](#), [335](#)

- data protection [334](#)
- Data Protection API (DPAPI) [316](#)
- debugging [434](#)
 - techniques [435-437](#)
 - tools [434](#)
- Decorator Pattern [206](#)
 - concepts [206](#)
 - concrete problem [206](#), [207](#)
 - implementation [207-209](#)
 - working [206](#)
- Denial of Service (DoS) [277](#)
- dependency injection
 - concepts [138](#), [139](#)
 - configuring, in ASP.NET Core [141](#), [142](#)
 - extending [154](#)
 - extension, need for [154-156](#)
 - implementing, in ASP.NET Core [140](#)
 - Inversion of Control [140](#)
 - service lifetimes [140](#)
- Dependency Inversion Principle (DIP) [25](#)
- dependency management [247](#)
- designing best practices, RESTful APIs
 - client side [106](#)
 - extension method refactoring [108](#), [109](#)
 - server side [108](#)
 - services [106](#), [107](#), [108](#)
- design patterns [43](#)
 - applications [43](#), [44](#)
 - Command Pattern [44-46](#)
 - component-based architecture [46](#), [47](#)
 - Option Pattern [52](#), [53](#)
- design principle-based architectures [63](#), [64](#)
- DI container [139](#)
- DI framework [139](#)
- Distributed Denial of Service (DDoS) [277](#)
- Docker Desktop
 - installing [362](#)
- Dockerfile [363](#)
- Domain-Driven Design (DDD) [14](#), [57](#), [72](#)
 - aggregates [75](#), [76](#)
 - bounded context [73](#)
 - domain events [78](#)
 - entities [73](#)
 - repository [76](#)
 - repository pattern [77](#), [78](#)
 - Ubiquitous Language [73](#)
 - value objects [73](#), [75](#)
- domain model [225](#)

Don't Repeat Yourself (DRY) principle [14](#), [28](#)

E

Elastic Stack (ELK) [356](#)

encryption [335](#), [336](#)

 implementing, with AES [336-338](#)

endpoint [114-116](#)

 building [116](#), [117](#)

end-to-end (E2E) tests [299](#)

Entity Framework Core (EF Core) [196](#)

 Database Context, creating [198-200](#)

 data model [197](#), [198](#)

 key features [197](#)

 using [197](#)

Error Notification System [347](#)

error tracking [346](#), [347](#)

EventCallback<T> [415](#), [416](#), [417](#)

event-driven architecture [62](#)

 components [63](#)

event-driven programming [49](#), [50](#)

 class, defining to manage events [50](#)

 event argument, defining [50](#)

 event handlers, wiring up [51](#)

 trigger events [51](#), [52](#)

event sourcing [223](#), [226](#)

exception handling [351](#)

 centralized error handling, with middleware [352](#), [353](#)

 custom error pages [353](#), [354](#)

 error, versus exceptions [351](#), [352](#)

 exception filters [354](#), [355](#)

Extending DI strategies [156](#)

 best practices [158](#)

 custom service factories [156](#)

 multiple instances, of same interface [157](#)

 options pattern [157](#)

F

Factory Method Pattern, with EF Core [201](#)

 concepts [201](#)

 concrete classes, creating [202](#)

 concrete problem [201](#)

 dependency injection, configuring [204](#), [205](#)

 Factory interface, defining [202](#)

 implementing [201](#), [202](#)

 in-memory-specific Factory class, creating [203](#), [204](#)

 SQLite-specific Factory class, creating [203](#)

 usage, in application [205](#), [206](#)

- working [201](#)
- functionality oriented techniques [184](#)
 - feature vertical slicing [186](#)
 - vertical slicing [184](#), [185](#), [186](#)
- Function as a Service (FaaS) [61](#)
- function file [297](#)

G

- Gang of Four (GoF) [14](#), [44](#), [191](#)
- gardening [166](#)
- GoF design patterns
 - application, in AS.NET Core [192](#), [193](#)
- Golden Hammer antipattern [165](#)
- Grafana [356](#)

H

- Handlers [228](#)
- HATEOAS [88](#)
- Hexagonal Architecture [79](#), [80](#)
 - application layer [81](#)
 - ASP.NET Core layer [83](#)
 - domain layer [80](#)
 - implementing [80-83](#)
 - infrastructure layer [82](#)
 - project structure [80](#)
- Hexagonal Architecture (HA) [57](#)
- horizontal layering [184](#)
- HTTP methods [92](#), [93](#)
- HTTPS [323](#)
- HTTP status codes [93](#), [94](#)

I

- input form validation [404](#), [405](#)
 - custom validation [409](#)
 - data annotations [405](#), [406](#)
 - implementing, in Blazor [406-409](#)
- integrated development environment (IDE) [3](#)
- integration testing [298](#), [299](#)
- interactive server-side rendering (interactive SSR) [34](#)
- Interface Segregation Principle (ISP) [22](#)
- Internet of Things (IoT) apps [2](#)
- inter-process communication (IPC) [181](#)
- Inversion of Control (IoC) [26](#), [140](#)

J

- JavaScript interop [418-424](#)
- JSON Web Token (JWT) [323](#), [330](#), [331](#)
 - using [331](#)
 - using, in ASP.NET Core authentication [331-333](#)

K

- Keep It Simple, Stupid (KISS) principle [14](#), [30](#)
- Kubernetes (K8s) [368](#), [369](#)
 - deployment files [369](#)
 - service files [369](#), [370](#)

L

- lava flow [166](#)
- Liskov Substitution Principle (LSP) [19](#)

M

- MapTaskListApi() [117](#)
- MessagePack security [277](#)
 - client configuration [278](#), [279](#)
 - custom resolver [279-282](#)
 - server configuration [278](#)
- MessagePackSecurity.UntrustedData [277](#)
- messages clients [271](#)
 - basic example, building [274](#), [275](#)
 - group client communication [272](#), [273](#)
 - MessagePack [273](#)
 - single client communication [272](#)
 - UI logic, adding [275](#), [276](#)
- microservices architecture [59](#), [360](#), [361](#)
 - drawbacks [60](#), [61](#)
 - features [60](#)
- middleware components [142-144](#)
 - common middleware scenarios [148](#)
 - in Program.cs [147](#), [148](#)
 - ordering [146](#), [147](#)
 - request pipeline, early termination [144](#), [145](#)
 - RUN delegate [145](#)
- middleware pipeline
 - building, with web application [145](#), [146](#)
- middlewares [38](#), [39](#)
 - built-in middleware [41-43](#)
 - recommended middleware [39](#), [40](#)
 - terminal middleware [40](#), [41](#)
- middleware scenarios
 - development environment [149](#)
 - production environment [149](#)

- staging environment [149](#)
- minimal APIs [101](#), [102](#)
 - structure [104-106](#)
- minimal APIs, with MVC [102](#)
 - controller [102](#), [103](#)
 - model [102](#)
 - views [103](#), [104](#)
- Model-View-Controller (MVC) [57](#)
 - model template [5](#)
- modular design
 - boundaries and responsibilities, defining [244](#)
 - principles [244](#)
- modularity [167](#), [168](#)
- modular monolith [239-241](#)
 - characteristics [241](#)
 - configuration [247](#)
 - deployment and updates [251](#)
 - detailed project structure [243](#)
 - logging and monitoring [249](#), [250](#)
 - project structure [242](#)
 - startup [247](#), [248](#)
 - testing [248](#), [249](#)
- module communication [244](#)
 - clear interfaces, defining [246](#)
 - code reviews [246](#), [247](#)
 - guidelines [246](#)
 - method calls [245](#)
 - module functionality, encapsulating [246](#)
 - service broker [245](#), [246](#)
 - shortcuts [246](#)
- monolithic architecture [58](#)
 - advantages [58](#), [59](#)
 - disadvantages [59](#)
- MVC architecture [64](#), [65](#)
 - implementing [66-68](#)

N

- NBomber [306](#)
- NET MAUI [395](#)

O

- Object-Relational Mapping (ORM) framework [196](#)
- Observer Pattern [209](#)
 - concepts [209](#)
 - concrete problem [210](#)
 - implementing [212-215](#)
 - working [209-211](#)

Open/Closed Principle (OCP) [17](#)
OpenTelemetry [356](#)
Option Pattern [52](#), [53](#)

P

parameters, in Blazor [410-415](#)
parameter transformer [123](#), [124](#)
performance tuning [440-443](#)
 daily tuning [444](#)
post-mortem analysis [348](#), [349](#)
post-mortem report [350](#), [351](#)
Progressive Web Apps (PWA) [376](#)
Prometheus [356](#)
Publisher/Subscriber pattern [62](#)

Q

query model [225](#)

R

Rapid Application Development (RAD) [171](#)
Razor component [387-389](#)
read model [225](#)
REpresentational State Transfer (REST) [87](#), [88](#)
resilience [170](#)
RESTful APIs [88](#)
 designing best practices [106](#)
REST principles [89](#)
 Client-Server architecture [89](#)
 statelessness [89](#)
 uniform interface [89-92](#)
route constraints [120](#)
 applying [122](#)
 constraint resolver [122](#)
route file-path template [120](#), [121](#)
route parameters [120](#), [121](#)
route templates
 catch-all route template [119](#), [120](#)
 defining [119](#)
routing [113-115](#)
 configuring [119](#)
 customizing [119](#)
 example [115](#), [116](#)
 in minimal APIs [117](#)
routing best practices [124](#)
 attribute routing, using [127-129](#)
 controllers and actions, organizing [131](#)

- missing routes, handling [132-134](#)
- RESTful routing, implementing [131](#), [132](#)
- route constraints, leveraging [129-131](#)
- simple and predictable routes [126](#), [127](#)
- versioning [124-126](#)

S

- scalability [168](#)
 - horizontal scalability [169](#)
 - vertical scalability [169](#)
- scaling out [169](#)
- scaling up [169](#)
- separation of concerns (SoC) [5](#)
- Serilog [319](#), [347](#), [356](#)
- serverless architecture [61](#)
 - benefits [62](#)
- server-sent events (SSE) [259](#)
- SignalR [254](#)
 - basic use [269](#)
 - client side SignalR [270](#), [271](#)
 - communication [267](#), [268](#)
 - communication server-client [254](#), [255](#)
 - configuring, at startup [271](#)
 - example use cases [268](#)
 - in .NET9 [269](#)
 - overview [266](#)
- SignalR application
 - MessagePack security [277](#)
 - securing [277](#)
- SignalR Hub
 - setting up [269](#), [270](#)
- single responsibility principle (SRP) [15](#), [164](#)
- Singleton design pattern [193](#)
 - concepts [193](#)
 - implementing [194-196](#)
 - working [193](#)
- software complexity [11](#), [12](#)
- software design principles [14](#)
 - Don't Repeat Yourself (DRY) [28-30](#)
 - Keep It Simple, Stupid (KISS) [30](#), [31](#)
 - SOLID principles [15](#)
- software system
 - building [171](#)
 - decision making, by ADRs [173](#)
 - decisions and trade-offs [173](#)
 - definition of requirements [172](#)
 - purpose and content [174](#)

- requirements analysis [172](#), [173](#)
- Software Termination Document (STD) [179](#)
- software testing methodologies
 - black-box testing [293](#)
 - V-Model testing [287](#), [288](#)
- SOLID principles [13](#), [15](#)
 - and architecture [186](#), [187](#)
 - Dependency Inversion Principle (DIP) [25-28](#)
 - Interface Segregation Principle (ISP) [22](#), [25](#)
 - Liskov Substitution Principle (LSP) [19-21](#)
 - Open/Closed Principle (OCP) [17-19](#)
 - single responsibility principle (SRP) [15](#), [16](#)
- SpecFlow [300](#), [303](#)
- SQL injection
 - protecting against [324](#), [325](#)
- SSL/TLS certificates [323](#)
- state management [424-430](#)
 - data storing, in memory [424](#)
- state management service
 - creating [424](#), [425](#)
- Strategy Pattern [215](#)
 - concepts [215](#)
 - concrete problem [216](#)
 - implementing [216-219](#)
 - working [215](#)

T

- TaskManagerApp component [47](#)
- TaskModel class [48](#), [49](#)
- technical debt [12](#)
- terminal middleware [41](#)
- termination criteria
 - definition [178](#), [179](#)
- testing, in modular monolith
 - end-to-end testing [249](#)
 - integration testing [249](#)
 - unit testing [248](#)
- testing strategies [437](#), [438](#)
 - best practices [439](#), [440](#)
 - continuous testing [439](#)
 - end-to-end testing [438](#)
 - integration testing [438](#)
 - performance testing [439](#)
 - unit testing [438](#)
- tests, in ASP.NET Core
 - acceptance testing [300-302](#)
 - acceptance testing, with SpecFlow [303](#), [305](#)

- Arrange-Act-Assert [295](#), [296](#)
- Behaviour-Driven Development (BDD) [296-298](#)
 - conducting [294](#)
 - end-to-end testing [299](#), [300](#)
 - integration testing [298](#), [299](#)
 - load testing [306](#)
 - results, interpreting [307](#), [308](#)
 - unit testing [295](#)
- Transport Layer Safety (TLS) [318](#)

U

- Ubiquitous Language [73](#)
- Unified Modelling Language (UML) [175](#), [289](#)
- Uniform Resource Identifiers (URIs) [89](#)
- URL mapping [118](#)
 - attribute routing [118](#), [119](#)
 - route parameters [118](#)
- Use Cases layer [183](#)
- User Acceptance Tests (UATs) [292](#)

V

- vertical slicing [184-186](#)
 - feature vertical slicing [186](#)
- Visual Studio (VS) [2](#)
- V-Model testing [287](#), [288](#)
 - architecture design [290](#)
 - coding [290](#), [291](#)
 - model design [290](#)
 - requirement design [289](#)
 - system design [289](#)
 - validation [291](#), [292](#)
 - verification [289](#)

W

- Waterfall methodology [171](#)
- WebAPI
 - securing [333](#), [334](#)
- Web APIs
 - containerizing [366-368](#)
- WebAssembly (Wasm) [384](#)
 - binary format [386](#)
 - execution [386](#)
 - structure [385](#)
 - text format [386](#)
 - validation [385](#)
- WebView2 [376](#)

write model [225](#)
wwwroot folder [37](#)

X

xUnit [301](#)
